Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Hochschule für Technik und Wirtschaft

Fachbereich Wirtschaftswissenschaften II

Studiengang Angewandte Informatik

Masterarbeit

Konzeption und Prototyp einer intelligenten Arbeitssuche nach persönlichen Fähigkeiten

---

vorgelegt von Sergej Meister
Matrikelnummer: s0521159

Erstgutachter: Prof. Dr. Christian Herta

Zweitgutachter: Prof. Dr.-Ing. habil. Dierk Langbein

Berlin, 3. März 2016

Vorwort

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Inhaltsverzeichnis

1. Einleitung

1.1 Motivation

Was versteht man unter dem Begriff *Information*? Was sind *Daten*? Die beiden Fragen sind sehr beliebt im Informatikstudium, besonders im ersten Semester. Doch die Informationen und Daten spielen nicht nur in der Informatik sondern auch in andren Gebieten eine wichtige Rolle. Täglich treffen die Menschen eine Entscheidung auf der Basis ihres Wissensdaten, aus denen Informationen extrahiert werden. Dadurch dass Informationen aus den Daten gelesen werden, hängt ein Entscheidungsfaktor wesentlich von der *„Qualität"* den bereitgestellten Daten ab.

In der heutigen Informationsgesellschaft durch das schnelle Wachstum von digital verfügbaren Daten ist es für die Menschen schwieriger geworden, die entscheidungsrelevante Informationen aus den Datenmengen zu filtern. Deswegen bietet das Internet viele verschiedene Suchportals für unterschiedliche Interessengruppe, die eine gezielte Suche nach bestimmten Datenmengen erlauben.

Die Jobportals unterstützen ihre Anwender rund um das Thema *„Arbeitsuche"* und *„Bewerbungsprozess"*. Ein Arbeitssuchender legt Suchkriterien, wie Berufsbeizeichnung und Region, fest und bekommt innerhalb weniger Sekunden eine Liste von Arbeitsangeboten, die den Suchkriterien entsprechen. Es ist offensichtlich, dass die Arbeitssuche dank Jobportals viel effizienter und einfacher geworden ist. Trotzdem muss der Arbeitssuchender immer noch viele Informationen selbst raus finden und interpretieren. Zum Beispiel für die Entscheidung, ob die Stelle den gewünschten Anforderungen entspricht, sind die Daten wichtig, wie Berufsname, Firmenname und Qualifikation. Für das Bewerbungsschreiben werden noch weitere Daten benötigt, wie Kontaktperson, Mailadresse, Firmenhomepage und Firmenadresse. Diese Informationen muss der Arbeitssuchender ständig selbst raus finden. Genau mit dieser Problematik beschäftigt sich das Programm *„IntelliJob"*, das diese Daten automatisch aus dem Arbeitsangebot extrahiert.

Das Programm *„Intellijob"* wurde im Rahmen der Veranstaltung *„Informationssystemen"* unter Betreuung von Professor Christian Herta entwickelt und kann bereits Berufsbezeichnung, Mailadresse, Firmenhomepage, Kontaktperson und Firmenadresse automatisch auslesen und in Datenbank speichern. Außerdem verfügt das Programm über eine Weboberfläche, um die gespeicherte Daten tabellarisch darzustellen.

Der weitere Nachteil von den Jobportals ist die Implementierung der Suche, die in der Profile gespeicherten personalisierten Daten nicht berücksichtigt. Denn die typische Suchkriterien eines Jobportals sind einfache Stichworte wie Softwareentwickler, Java, C♯ u.s.w. Daraus wird es ersichtlich, dass die Suche nach einem konkreten Beruf erfolgt. Dieses Suchverfahren eignet sich gut für Quereinsteiger und für die Menschen, die auf der Suche nach neuer Berufsorientierung sind, aber nicht für die Menschen, die eine Arbeit entsprechend ihren Fähigkeiten und Qualifikationen suchen. Wäre es nicht besser, wenn die Suche nach persönlichen Fähigkeiten und Qualifikationen erfolgte, die von Benutzer selbst bewertet werden? Nehmen wir an, der Mensch bewertet seine Java-Kenntnisse mit 2 und C♯ mit 5 Sternen. Dann ist es doch logisch, dass alle C♯ Arbeitsangebote in der Ergebnisliste zuerst angezeigt werden müssen. Vielleicht ist es auf dem ersten Blick nicht ganz ersichtlich, woran die beide Verfahren sich unterscheiden. Die Suche nach einem Beruf als Softwareentwickler kann aber die Java- Arbeitsangebote zuerst anzeigen, was aus der Sicht eines Arbeitssuchender mit den besseren C♯ Kenntnissen nicht ganz zutreffend ist. Der weitere Unterschied liegt im Entscheidungsfaktor. Bei dem ersten Verfahren entscheidet der Mensch, welche Arbeit er haben will. Bei dem zweiten Verfahren entscheidet das Programm, welche Arbeit für den Mensch am besten geeignet ist.

Die Untersuchung von mehreren Jobportals wie *„Xing*", *„StepStone*", *„JobScout24*", *„Monster*" hat bestätigt, dass es tatsächlich kein deutsches Jobportal gibt, das die Suche nach persönlichen Fähigkeiten und Qualifikationen anbietet.

1.2 Ziele der Arbeit

Im Rahmen dieser Masterarbeit wird ein Suchverfahren implementiert, dass die Suche nach persönlichen Fähigkeiten und Qualifikationen unterstützt. Die Suche muss mit *Elasticsearch*[1] realisiert und in die Anwendung *IntelliJob* integriert werden. Um die Suchqualität bewerten zu können, müssen mehrere unterschiedliche Suchanfragen gebildet und genau untersucht werden. Am Ende der Arbeit wird geprüft, ob das neue Suchverfahren gegenüber der traditionellen Suche nach Stichworte ein besseres Ergebnis liefert. Das Suchverfahren wird als effektiv bewertet, wenn die erste Arbeitsangebote in der Ergebnisliste den Benutzeranforderungen besser entsprechen werden.

Das weitere Thema dieser Arbeit wird die Datenbeschaffung sein. Um nach persönlichen Fähigkeiten und Qualifikationen suchen zu können, müssen diese Daten zuerst erfasst werden. Dafür wird das Projekt *„IntelliJob*" um eine neue Web-Form erweitert, die die Bewertung von persönlichen Fähigkeiten und Qualifikationen ermöglicht. Die Daten, die dem Arbeitssuchender bereitgestellt werden, müssen sinnvoll gruppiert werden, so dass die Kriterien wie Java und C♯ unter den Sammelbegriffen Programmieren oder Softwareentwickeln gefunden werden könnten. Der Arbeitssuchender muss in der Lage sein, seine eigene Daten speichern, editieren

[1] **www.elastic.co**.

und löschen zu können. Außerdem darf der Benutzer selbst entscheiden, welches Suchverfahren eingesetzt werden soll.

1.3 Aufbau der Arbeit

Der schriftliche Teil der Arbeit beschreibt ein kompletter Entwicklungsprozess. Im Kapitel *„Grundlagen"* werden die Technologie erläutert, die für die Entwicklung neues Suchverfahrens relevant sind. Dabei wird das Projekt *IntelliJob* und die eingesetzte Technologie kurz vorgestellt. Das Kapitel *„Systementwurf"* befasst sich mit der Softwarearchitektur und beschreibt die Logikverteilung und die Datenkommunikation innerhalb der Anwendung. Das vierte Kapitel *„Entwicklung und Implementierung"* stellt das Prototyp des neuen Suchverfahrens vor. Dabei wird sowohl das Datenmodel als auch die wichtigste Methoden detailliert beschrieben. Das Kapitel besteht aus mehrere Unterkapiteln. Jedes Unterkapitel beschreibt die Implementierung von einem konkreten Anwendungsfall. Das Kapitel *„Evaluierung"* analysiert mehrere unterschiedliche Suchanfragen und bewertet sie. Problemen und Schwierigkeiten, die während der Implementierung aufgetreten sind, sowie die Art und Weise, wie diese Probleme gelöst oder eventuell nicht gelöst wurden, werden in dem 6. Kapitel erfasst. Zum Schluss wird die Arbeit zusammengefasst und ein Resümee gezogen. Ein Ausblick zeigt auf, welche Erweiterungen für die Anwendung in Zukunft noch sinnvoll sein könnten.

2. Grundlagen

In diesem Kapitel wird auf die allgemeinen und informatischen Grundlagen eingegangen, welche zum Verständnis dieser Masterarbeit relevant sind. Die Anwendung *IntelliJob* persistiert bereits die Daten in eine dokumentenorientierte Datenbank *MongoDB*[1] und es wird noch eine weitere dokumentenorientierte Technologie *Elasticsearch* eingesetzt. Deswegen ist es wichtig mindestens die Grundlagen sowie die Vor- und Nachteile von dokumentenorientierten Datenbanken im Blick zu behalten. Die in der Arbeit verwendete *Elasticsearch-Feature's* werden in einem extra Kapitel ausführlich beschrieben. Da die Suchfunktionalität in die bereits vorhandene Anwendung integriert werden muss, ist es für die Bewertung der Arbeit notwendig den IST-Zustand zu dokumentieren.

2.1 Dokumentenorientierte Datenbanken

Bei dokumentenorientierten Datenbanken werden die Daten in Form von Dokumenten gespeichert. Die typische Dokument-Formate sind XML, YAML und JSON. Die einzelnen Datenfelder werden als Key-Value Stores zusammengefasst. Jedes Dokument ist vollkommen frei bezüglich seine Struktur und das ist ein wesentlicher Unterschied zu relationalen Datenbanken, wo die Datenstruktur vordefiniert ist [Dorschel 2015]. Die Abbildung 2.1 zeigt eine typische normalisierte Datenstruktur in der relationalen Welt. Einige Datensätze in der Tabelle *skills* beinhalten kein *"description"*. Die Spalte muss in dem Fall einfach leer bleiben. Es besteht keine Möglichkeit nur für bestimmte Datensätze auf die Spalte *"description"* zu verzichten.

category_id	name
1	Languages
2	Knowledges
3	Personal strengths

skill_id	category_id	name	description
1	1	German	
2	1	English	
3	2	Java	programming language java
4	2	PHP	programming language php

Abbildung 2.1: Relationale Tabellen *skill-categories* und *skills*

In dokumentenorientierten Datenbanken ist es aber durchaus möglich, dass einige Dokumente innerhalb einer *Collection* kein *description* Feld haben. Das sieht man deutlich in der *Collection skills* auf der nächsten Seite.

[1] **www.mongodb.org**.

Mongo Collection *skill-categories*

```
{
  "_id": "569ab65644ae6344028b6cb7",
  "name": "Languages"
},
{
  "_id": "569ab65644ae6344028b6cb7",
  "name": "Knowledges"
},
{
  "_id": "569ab66d44ae6344028b6cb9",
  "name": "Personal strengths"
}
```

Mongo Collection *skills*

```
{
  "_id": "569ab85144ae6344028b6cbf",
  "category": {
    "_id": "569ab61a44ae6344028b6cb5",
    "name": "Languages"
  },
  "skills": [
    {
      "_id": "569ab8dd44ae6344028b6cc0",
      "name": "deutsch"
    },
    {
      "_id": "569ab8dd44ae6344028b6cc1",
      "name": "english"
    }
  ]
},
{
  "_id": "569ab9d644ae6344028b6cc9",
  "category": {
    "_id": "569ab65644ae6344028b6cb7",
    "name": "Knowledges"
  },
  "skills": [
    {
      "_id": "569ab9d644ae6344028b6cca",
      "name": "java",
      "description": "programming language java"
    }
  ]
}
```

Wenn man die beide *Collection* genau anschaut, fehlt es sofort auf, dass die *Category-Daten* nicht redundant sind. In den relationalen Datenbanken werden die Datenattribute durch die Anwendung von verschiedenen Normalisierungsregeln solange zerlegt, bis keine vermeidbaren Redundanzen mehr vorhanden sind. Die dokumentenorientierte Datenbanken verfolgen den Prinzip der *Aggregation*. Der Begriff kommt eigentlich aus dem Domain-Driven Design und bedeutet, dass jedes Dokument eine eigenständige Einheit ist, das von keinen anderen beziehungsweise von möglichst wenigen Dokumenten abhängig ist. Dadurch können die Daten immer vollständig automar gespeichert oder gelesen werden.

Diese hohe Datenflexibilität ist aber nicht umsonst. Zum Ersten ist es oft aufwendig die Daten zu ändern, die bereits in anderen Dokumenten gespeichert sind. Ein Beispiel wäre dafür das Dokument mit dem Name *„Languages"* in der *Collection „skill-categories"* mit einem neuen Attribut zu erweitern oder komplett zu löschen. In dem Fall muss diese *„Category"* auch in allen anderen Dokumenten gefunden und entsprechend geändert werden. Zum Zweiten können dokumentenorientierte Datenbanken aufgrund ihrer Schemafreiheit keine einfache Datenvalidierung der Attributen und Datentypen durchführen. Das muss stattdessen im Programmcode behandelt werden, was zu höheren Softwarekosten führt und fehleranfällig ist [Jan 2013].

Wenn man das ganze kurz wörtlich zusammenfasst: Schemafreiheit, keine Datenvalidierung, mehr Programmcode, Fehleranfällig und aufwändige Datenänderung auf der Datenbankebene, fragt man sich, wieso werden aber die dokumentenorientierte Datenbanken, die seit 2008 unter dem Titel *„NoSQL"*[2] bekannt sind, immer beliebter? Eine einfache Antwort lautet, dass die *NoSQL*-Datenbanken oft schneller als relationale Datenbanken sind. Statt mehrere Tabellen mit *JOIN*-Operation abfragen zu müssen, führt ein *NoSQL*-Datenbank nur eine einzige Abfrage aus, um ein kompletter Datensatz zu erhalten. Die Datenkonsistenz beim Lesen und Schreiben in mehrere Tabellen wird in klassischen relationalen Datenbanken durch Transaktion-Mechanismen realisiert. Jede Transaktion ist isoliert von einander und wird ganz oder gar nicht ausgeführt. Wenn eine Transaktion in eine Tabelle schreibt, kann keine andere Transaktion auf die Tabelle zugreifen, bis der Schreibvorgang komplett abgeschlossen oder unterbrochen wird. Das führt dazu, dass die Lese- und Schreiboperation von parallel ausgeführten Transaktionen einander blockieren und warten müssen, bis alle nötige Ressource freigegeben werden. Dadurch dass *NoSQL* Datenbanken keine Abfrage blockieren, werden die auch schneller abgearbeitet. Es besteht zwar die Möglichkeit, dass inkonsistente Daten gelesen werden, aber aufgrund, dass die Daten oft nur aus einer einzigen *Collection* gelesen werden, ist die Fehlerwahrscheinlichkeit enorm klein im Vergleich zu viel steigender Performance[Dorschel 2015]. Die einzige Schutzmaßname, die von meisten NoSQL-Datenbanken unterstützt wird, ist die Datenversionierung. Bei jedem Speicherzugriff wird die Version des Dokumentes geprüft, ist die Version identisch dem bereits gespeicherten Dokument, dann wird ein *Exception* geworfen. Denn es ist ein Zeichen dafür, dass Dokument bereits von einem anderen *Request* geändert

[2] **https://de.wikipedia.org/wiki/NoSQL**.

wurde. Außerdem wird auch die Datenversionierung für die Synchronisierung zwischen mehrere Rechnerknoten innerhalb eines *Clusters* verwendet. Das *Sharding Concepts* in dokumentenorientierten Datenbanken wird im Kapitel über *Elasticsearch* detailliert beschrieben.

Fazit: Die dokumentenorientierte Datenbanken eignen sich besonders gut für die Anwendungen, wo hohe Geschwindigkeit bei Datenzugriffe und Datenverarbeitung im Vordergrund steht und die Datenredundanz nicht hoch priorisiert wird. Die meisten Datenbankzugriffe in der Anwendung *IntelliJob* sind Lesezugriffe. Es gibt wenige Möglichkeiten die Daten durch Benutzeraktion zu ändern. Die Änderungen beeinflussen oft nur ein einziges *Collection* und meistens sogar nur ein einziges Dokument. Deswegen kann eine dokumentenorientierte Datenbank die Anwendungsanforderungen viel besser erfüllen.

2.2 IntelliJob

Die Anwendung *„IntelliJob"* wurde bereits mehrmals erwähnt und muss endlich vorgestellt werden. Es gab mal eine Idee die bewerbungsrelevante Daten aus Arbeitsangeboten automatisiert zu extrahieren. Diese Idee wurde mit dem Programm *„IntelliJob"* realisiert. Da die Anwendung im Rahmen der Veranstaltung *„Informationssystemen"* eigenständig entwickelt und dokumentiert wurde, werden einige Inhalte aus der Arbeit in diesem Kapitel zitiert.

Bevor die Daten aus dem Arbeitsangebot gelesen werden könnten, muss dieses Arbeitsangebot erstmals ermittelt werden. Dafür werden mehrere Job-Agenten auf Jobportals *Stepstone*[3] und *Monster*[4] angelegt, die die Arbeitsangebote per Email täglich senden. Das Email beinhaltet mehrere HTML-Links mit unterschiedlichen Arbeitsangeboten. Die Anwendung *„IntelliJob"* holt diese Emails aus dem Postfach ab, ruft die Links auf und speichert das geladene Inhalt in Datenbank ab.

Im nächsten Schritt werden die bewerbungsrelevante Daten, *Berufsbezeichnung*, *Firmenadresse*, *Firmenhomepage*, *Mailadresse* und *Kontaktperson*, aus dem Arbeitsangebot extrahiert.

Berufsbezeichnung wird einfach aus dem HTML-Link ausgelesen, das als *Value* immer eine Berufsbezeichnung enthält.

Mailadresse und **Firmenhomepage** werden mit Hilfe von Regulären Ausdrücken extrahiert.

Mail-Pattern: *[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+.[a-zA-Z0-9-.]+*

WWW-Pattern: *www.[\w\d.:#@%/;$()~_?+-=&]**

HTTP-Pattern: *http.[\w\d.:#@%/;$()~_?+-=&]**

[3] **www.stepstone.de**.
[4] **www.monster.de**.

HTTPS-Pattern: *https.[\w\d.:#@%/;$()~_?+-=&]**

Leider können Reguläre Ausdrücke nur Daten mit einer eindeutigen Struktur ermitteln. **Kontaktperson** und **Firmenadresse** können aber sehr unterschiedlich geschrieben werden:

- Alle der Kosmonauten 26a D-32451 Bad Kreuznach
- Straße des 17. Juni 17a 13456 Berlin (Lieblings Beispiel)
- Herr Richard-Alexander Wagner
- Johann Wolfgang von Goethe
- Alexander Sergejewitsch Puschkin

Aus den oben aufgelisteten Beispielen wird es ersichtlich, dass es kaum möglich ist, allein mit Regulären Ausdrücke, diese Daten zu extrahieren. Deswegen werden diese Daten in *„IntelliJob"* mit Hilfe von *„Apache OpenNLP Framework"*[5] ermittelt und zusammen mit allen anderen bewerbungsrelevanten Daten in Datenbank gespeichert.

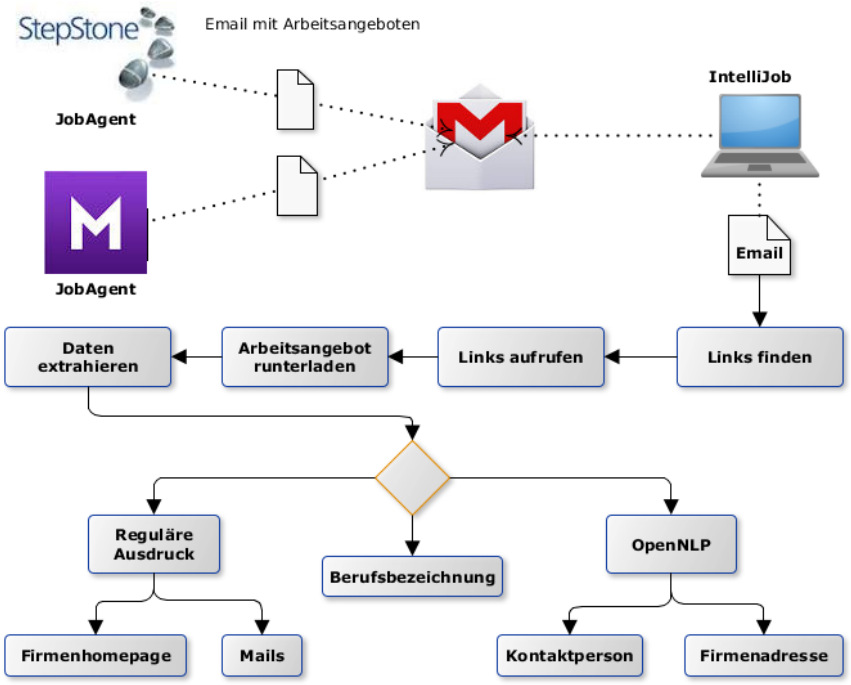

Abbildung 2.2: Prozessablauf in IntelliJob

[5] **https://opennlp.apache.org**.

2.2.1 Technologie Stack

Apache OpenNLP ist ein Open Source Produkt und wird als Maven-Dependency in der Version 1.5.3 in das Projekt *„IntelliJob"* eingebunden. Das Framework setzt die Technologie aus dem Gebiet *Natural Language Processing* ein, um bestimmte Bestandteile eines Textes zu erkennen und zu klassifizieren. Die Muster-Erkennung basiert auf künstlich erzeugten Trainingsdaten, die durch ein Lernprozess in einem Model gespeichert werden. Das Framework wird ständig weiter entwickelt und verfügt über eine gute Dokumentation.[6]

Die Anwendungslogik von *„IntelliJob"* wird in der Programmiersprache Java geschrieben und verwendet sehr viele Komponenten aus dem Spring Framework[7]. Spring Framework ist ebenfalls ein Open Source Produkt, welches performante und moderne Java-Enterprise Einsätze realisiert. Mit dem Einsatz von Spring *Dependency Injection* und *Aspekt orientierte Programmierung* wird das Programmcode redundant verteilt und kann auch leicht wiederverwendet werden. Alle erzeugte Objekte werden anhand der *Spring Data*[8] in eine dokumentenorientierte Datenbank MongoDB persistiert.

Die weitere Besonderheit von *„IntelliJob"* ist, dass es eine Standalone Web-Anwendung ist, die lokal auf dem Client einen embedded Web Container *Tomcat*[9] startet. Der große Vorteil von diesem Einsatz liegt darin, dass der Webserver ein Teil der Anwendung ist, und die Anwendung nicht mehr in den Webserver deployed werden muss. Das Programm wird als JAR-Datei ausgeliefert und kann direkt in der Java Virtual Machine ausgeführt werden. Realisiert wird das ganze mit *Spring Boot*[10] Technologie, die dank Autokonfiguration sowohl alle nötige Resource zusammenpackt, als auch alle Rest-Services zur Kommunikation mit Frontend initialisiert.

Das Frontend wird mit *AngualrJS*[11] umgesetzt und verfügt bereits über 6 verschiedene Views. Das Framework unterstützt die Entwickler bei der Implementierung von Single Page Anwendungen. Während der Masterarbeit wird *AngularJS* von der Version 1.2.16 zu 1.4.7 aktualisiert.

Home	Startseite
Emails	zeigt alle gefundene Emails von Job Portals
Job Links	zeigt alle in Mails gefundene Links zu Arbeitsangeboten
Jobs	zeigt alle runter geladene Arbeitsangebote
Jobs Details	zeigt alle extrahierte Daten eines Arbeitsangebotes
Audit	dient zur Datenauswertung und zeigt sowohl die aktuellste als auch die alte Ergebnisse des Datenextraktions.

[6] **https://opennlp.apache.org/documentation/1.6.0/manual/opennlp.html**.
[7] **www.spring.io**.
[8] **http://projects.spring.io/spring-data**.
[9] **http://tomcat.apache.org**.
[10] **http://projects.spring.io/spring-boot**.
[11] **https://angularjs.org**.

2.2.2 Software Architektur

„IntelliJob" ist quelloffen und kann auf Versionsverwaltungsplattform, GitHub[12], heruntergeladen werden. Die Anwendungslogik basiert auf Schichtenarchitektur, die sequenziell von oben nach unten abgearbeitet wird.

Präsentationsschicht repräsentiert die Daten in HTML-Form und regelt die Interaktionen zwischen Benutzer und Software. Alle Benutzeranfragen werden asynchron an Serviceschicht weitergeleitet. Die Daten zwischen beiden Schichten werden in JSON-Format übertragen.

Serviceschicht nimmt die Client-Anfragen entgegen, validiert Benutzerdaten und wandelt sie in Domain-Objekte um. Danach wird die Abfrage der Controlschicht übergeben. Letztendlich wird ein *Response* generiert und zurückgesendet.

Controlschicht vereint alle Methoden aus Datenzugriffsschicht und führt alle nötige Operationen aus, um die Benutzeranfrage zu bearbeiten.

Datenzugriffsschicht kapselt die Zugriffe auf persistente Daten. Die Datenaustausch erfolgt über Domain-Objekte.

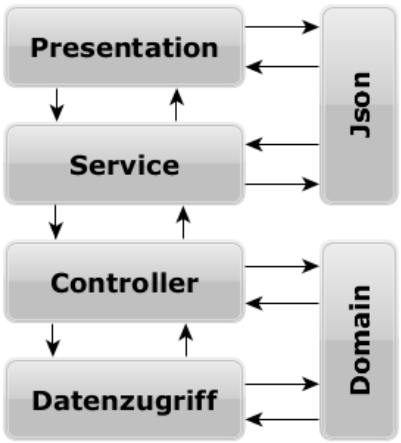

Abbildung 2.3: Schichtenarchitektur

Dadurch dass die Anwendungslogik in mehrere Schichten verteilt wird, wird die Komplexität der Anwendung wesentlich reduziert, was sowohl für das Verständnis als auch für die Wartung von Software ein großer Vorteil ist. Außerdem können die Programmteile dank ihrer Abstraktion gut getestet und auch leicht wiederverwendet werden.

Um die Methoden zum Datenextraktion auch in anderen Projekten verwenden zu können, werden sie in ein separates Projekt, namens *Civis-Tools*[13], ausgelagert.

[12] **https://github.com/SergejMeister/intellijob**.
[13] **https://github.com/SergejMeister/civis-tools**.

2.2.3 Problembeschreibung

Zum Beginn der Masterarbeit gab es ein kleines Problem in *„IntelliJob"*, das früher immer missachtet wurde und während der Masterarbeit unbedingt gelöst werden müsste. Das Problem liegt in doppelt gespeicherten Arbeitsangeboten.

Jedes Mail von Jobportal beinhaltet Links sowohl mit neuen als auch mit alten Arbeitsangeboten. Deswegen wird es vor dem Schreiben in Datenbank immer geprüft, ob das Link zu diesem Arbeitsangebot schon früher gespeichert wurde. Durch die Beobachtung wurde allerdings erkannt, dass die Links zum gleichen Arbeitsangebot manchmal geändert werden. Außerdem kann ein Arbeitsangebot sowohl von *StepStone* als auch von *Monster* Portal kommen. Die Links und Mail sind dabei unterschiedlich und es gibt auch keine logische Möglichkeit die beiden Arbeitsangebote von einander zu unterscheiden.

2.3 Tag Cloud

Abbildung 2.4: Tag Clouds - *www.einfachbewusst.de*

Die Tag Cloud, deutscher Begriff Schlagwortwolke, werden im Web immer beliebter und dienen zur Visualisierung von gewichteten Daten. Je nach Gewichtung werden die Daten in verschiedenen Schriftgröße und Farben dargestellt. Mit Hilfe von Tag Cloud können beispielsweise die meistbesuchte Beitrage eines Weblogs oder die häufigste Worte eines Dokuments angezeigt werden. In der Implementierungsphase dieser Masterarbeit werden die Tag Clouds zur Visualisierung von persönlichen Fähigkeiten in der Suchmaske eingesetzt. Dadurch wird es für den Benutzer ersichtlich, welche Fähigkeiten bei der Suche besonders priorisiert werden.

In der Praxis werden die einzelne Wörter der Tag Cloud mit einer Unterseite oder Dokument verlinkt. So entsteht eine Art alternative zu einem Navigationsmenü. Der Nachteil von diesem Einsatz besteht darin, dass die Gewichtung der Schlagwörter durch Aufruf der Seite oder Dokumentes erfolgt und keine Information darüber enthält, ob der verlinkte Inhalt den Benutzererwartungen entspricht. Der Benutzer klickt auf dem hoch gewichteten Schlagwort an, stellt fest, dass der verlinkte Inhalt nicht den Erwartungen entspricht und verlässt die Seite. Da die Seite jedoch

aufgerufen wurde, wird der Schlagwort der Tag Cloud höher gewichtet[OnPage 2016].

Die Tag Cloud wird meistens nach der folgenden Formel berechnet:

$$S_i = [(f_{max} - f_{min}) * \frac{t_i - t_{min}}{t_{max} - t_{min}} + fmin]$$

- S_i - anzuzeigende Schriftgröße
- f_{max} - maximale Schriftgröße
- f_{min} - minimale Schriftgröße
- t_i - Häufigkeit des betreffenden Schlagwortes
- t_{min} - Häufigkeit, ab der ein Schlagwort angezeigt werden soll
- t_{max} - Häufigkeit des häufigsten Schlagwortes[14]

2.4 Informationsbeschaffung

Im ersten Kapitel wurde schon erwähnt, dass die Informationen aus Daten extrahiert werden. Zur Informationsbeschaffung (eng. Information Retrieval) gehören neben Extraktionsprozess noch weitere Prozesse. Zuerst müssen die Daten erfasst werden und in einer Datenbank oder einem Daten-Management-System in einer bestimmen Syntax abgelegt werden. Dabei legt der Verfasser einen Teil seines Wissen in ein Dokument nieder, das in Text, Bild oder eine andere Form gespeichert wird. Im nächsten Schritt müssen die Daten, die bereits existieren, wiedergefunden und in eine geeignete Repräsentationsform gebracht werden. Letztendlich sollen nur die Informationen extrahiert werden, die möglichst gut bei einer Problemlösung helfen können. Der Autor Andreas Heinrich in seinem Lehrtext *„Information Retrieval Grundlagen, Modelle und Anwendungen*" bezeichnet dieses Prozess als Wissenstransfers und weist deutlich darauf hin: *„... Information Retrieval nicht nur als den Prozess der Wissensextraktion aus einer Dokumentenkollektion sondern als Prozess des Wissenstransfers zu verstehen*"[Henrich 2008]. Die Abbildung 2.7 stellt das Prozess vereinfacht dar.

Das System, das sich mit der Informationsbeschaffung befasst, wird in der Informatik als IR-System bezeichnet. Die Aufgabe des IR-Systems besteht hauptsächlich darin, die hinter der Anfrage stehende Informationsbedürfnisse zu befriedigen. Oft muss ein IR-System neben der Anfrage noch weitere Datenquellen, wie Feedback oder vorhandene Information über Nachfragender, zu berücksichtigen, um eine bessere Informationsergebnis zu erzielen. Im Bezug auf die Anwendung *„IntelliJob*" kann Informationsergebnis wesentlich verbessert werden, indem der Nachfragender die Suchanfrage selber priorisiert.

[14] **https://de.wikipedia.org/wiki/Schlagwortwolke**.

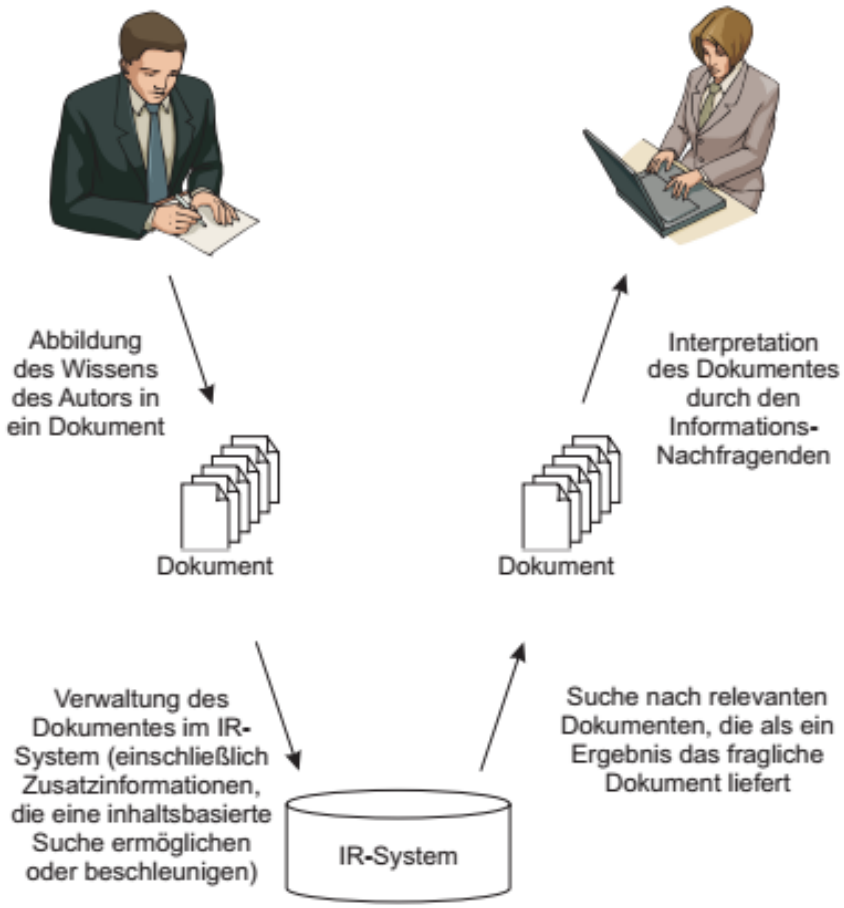

Abbildung 2.5: Prozess des Wissenstransfers vom Autor eines Dokumentes bis zum Informations-Nachfragenden

2.4.1 Qualität in Information Retrieval System

Um ein gutes IR-System von einem schlechten unterscheiden zu können, muss die Qualität der Ergebnissen gemessen werden. Die Qualität wird ihrerseits durch **Relevanz**, **Recall** und **Precision** bestimmt.

Relevanz kann binär (*relevant/nicht relevant*) oder mit Relevanzeinstufung (*von 1 bis 10*) definiert werden. In der Praxis hat sich allerdings die binäre Relevanz durchgesetzt. Die Entscheidung, ob ein Dokument relevant oder nicht relevant ist, ist sehr subjektiv und erfordert Vorwissen sowohl über das untersuchte Dokument als auch über andere Dokumenten. Deswegen kann es sehr schwierig sein alle Dokumente einzustufen[Henrich 2008].

Recall beschreibt der Anteil der relevanten Dokumenten in der Ergebnismenge im Verhältnis zu allen relevanten Dokumenten und wird wie folgt berechnet:

$$Recall = \frac{a}{S}$$

- a - Anzahl der relevanten Dokumente im Ergebnis

- S - Gesamtzahl der relevanten Dokumente

Precision zeigt die Genauigkeit der Ergebnismenge.

$$Precision = \frac{a}{D}$$

- a - Anzahl der relevanten Dokumente im Ergebnis
- D - Gesamtzahl der Dokumente im Ergebnis

Die oben beschriebene Vorgehensweise zur Bestimmung von *Recall* and *Precision* Werten eignet sich gut für unsortierte Ergebnismenge, wo es hauptsächlich darum geht möglichst viele relevante Dokumente zu finden. In der heutigen Informationsgesellschaft kann eine Ergebnisliste extrem viele Dokumente beinhalten und alle diese Dokumente können auch unterschiedlich relevant sein. Am Interessantesten sind deswegen nur die erste k-Dokumente der Ergebnisliste. Für die Evaluierung von sortierter Liste, auch Ranking Liste genannt, wird eine Folge von *Recall* und *Precision* Werten betrachtet, die für jedes Dokument einzeln berechnet werden. Die Vorgehensweise ist folgendes:

- Zuerst muss eine Testmenge mit k-Dokumenten zur Evaluierung festgelegt werden.
- Wenn ein Dokument aus der Testmenge in der Ergebnisliste gefunden wird, werden die *Recall* und *Precision* Werte berechnet. Grundsätzlich muss es auch gelten: $Recall(k+1) > Recall(k)$ und $Precision(k+1) > Precision(k)$
- Wenn ein Dokument nicht gefunden wird, bleibt die *Recall* Wert unverändert. Die *Precision* Wert muss aber kleiner werden: $Recall(k+1) = Recall(k)$ und $Precision(k+1) < Precision(k)$

Laut der Definition kann jetzt für einen *Recall* Wert mehrere *Precision* Werte geben. Um es linear auszugleichen, wird der höchsten *Precision* Wert für jeden *Recall* Wert genommen. Das wird als *interpolated precision* bezeichnet[Manning u. a. 2008, S.158-159]

2.4.2 Indexierung von Textdokumenten

Die vorliegende Arbeit beschäftigt sich mit Arbeitsangeboten, die in Textform vorliegen. Diese Textdokumente müssen automatisiert erfasst werden. Dieses Prozess wird in Information Retrieval als *Indexierung* bezeichnet. Bei der Indexierung wird ein Dokument analysiert und alle Wörter außer *Stopwörter* werden ermittelt. Die *Stopwörter* sind besondere Wörter die sehr häufig vorkommen. In der deutschen Sprache sind Artikeln ein typisches Beispiel für die *Stopwörter*. Die extrahierte Worte werden in Grundform (z.B. *schreiben* - *schreib*) gebracht und als Terme abgespeichert. Durch Grundformredaktion wird es verhindert, dass die gleiche aber unterschiedlich geschriebene Worte mehrmals gezählt werden. Denn während der

Indizierung wird sowohl das Vorkommen eines Wortes in einem Dokument(term frequency, TF) als auch in allen anderen Dokumenten (inverse document frequency, IDF) gezählt.

$$IDF_i = log\frac{N}{n_i}$$

- i - der untersuchte Term
- N - Anzahl aller Dokumenten
- n_i - Anzahl der Dokumente mit Term

Der Logarithmus sorgt dafür, dass ein oft vorkommender Term nicht extrem hoch gewichtet wird. Das Endergebnis der automatischen Indexierung ist ein Gewicht für jeden Term in jedem Dokument[Mandl 2001]. Nachdem das Dokument erfolgreich indexiert wurde, kann es auch durchgesucht werden.

2.4.3 Boolesche Suche

Die Boolesche Suche oder das Boolesche Retrieval Modell basiert auf die elementare Mengenoperationen, wie Schnittmenge (AND-Verknüpfung), Vereinigungsmenge (OR-Verknüpfung) und Komplementärmenge (NOT-Verknüpfung). Das Modell geht davon aus, dass das Ergebnis einer Suchanfrage als exakte Menge von relevanten Dokumenten bestimmt werden kann. Dabei werden nur Dokumente ausgegeben, die den in der Suchanfrage formulierten booleschen Ausdruck erfüllen.

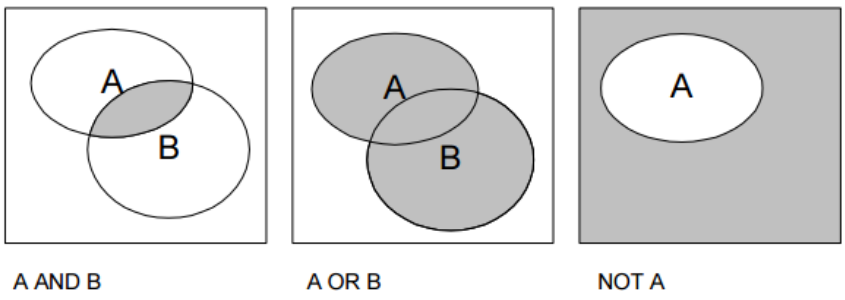

Abbildung 2.6: Die elementaren booleschen Operationen

Der Nachteil des booleschen Modells besteht darin, dass die Gewichtung von Termen nicht berücksichtigt wird und die Ergebnismenge deswegen ungeordnet ausgeliefert wird. Außerdem werden viele Dokumente nicht gefunden, weil die gesuchte Elemente nicht in den exakt gleichen Form in Dokumenten enthaltet sind. Trotz seiner Ungenauigkeit wird das boolesche Model oft eingesetzt vor allem um mehrere Suchanfragen mit einander zu verknüpfen.

2.4.4 Vektorraummodell

Zum ersten Mal wurde das Vektorraummodell von Gerard Salton, Andrew Wong und Chungshu Yang in einem Zeitschriftartikel unter dem Thema *„A Vector Space*

Model for Automatic Indexing" im Jahr 1975 publiziert und ausführlich beschrieben[Salton u. a. 1975]. Die Grundidee dieses Modells besteht darin, dass sowohl die Anfrage als auch alle Dokumente in einem vieldimensionalen Vektorraum dargestellt werden. Dabei bildet jeder einzelne Term eine Achse beziehungsweise eine Dimension in diesem Vektorraum und die Koordinaten werden durch die Gewichtung von Termen repräsentiert.

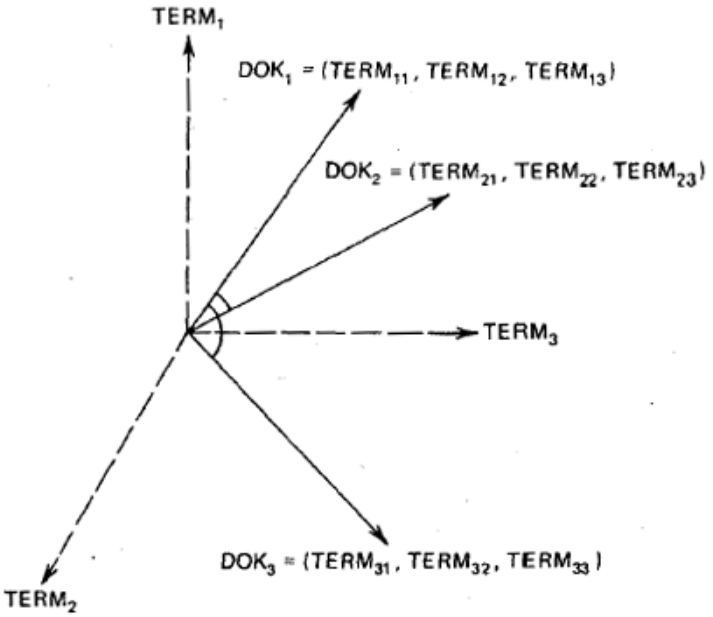

Abbildung 2.7: Verktorraumpräsentation eines Dokumentraums (Salton u. McGill 1987, 129)

Die Relevanz eines Dokumentes im Vektorraummodell wird oft durch Berechnung des Cosinus des Winkels zwischen Anfrage und Dokument bestimmt. Das führt dazu, dass nicht mehr nach exakten Übereinstimmungen sondern nach der Ähnlichkeit zwischen Suchanfrage und Dokumenten gesucht wird und die Ergebnismenge in dem Fall auch sortiert ausgegeben werden kann.

Obwohl das Vektorraummodell die wesentliche Nachteile des booleschen Modells aufhebt, kann das boolesche Model unter bestimmten Bedienungen eine bessere Ergebnismenge als Vektorraummodell liefern. Damit die Ähnlichkeitsmaß zwischen Anfrage und Dokument mit dem Vektorraummodell sinnvoll berechnet werden könnte, muss die Anfrage aus mehrere Begriffe bestehen. Wenn die Suchanfrage aus wenigen Begriffen besteht, kann das boolesche Model, das diese Begriffe einfach mit *AND* verknüpft, oft zu besseren Ergebnissen führen.

2.4.5 Volltextsuche

Bei der Volltextsuche handelt es sich darum, eine benutzerdefinierte und strukturierte Suchanfrage in einem unstrukturierten Textdokument zu finden. Natürlich kann man dafür das boolesche oder Vektorraummodell einsetzen, um die erste

Ergebnisse schnell zu bekommen. Allerdings bieten die meisten IR-System noch einige Optimierungsmethode, die die Ergebnismenge wesentlich beeinflussen können. Der nachfolgende Abschnitt beschreibt kurz die Optimierungsmethode, die für den praktischen Teil nützlich sein könnten.

Phrasensuche wird für die Suche nach zusammenhängenden Begriffen eingesetzt. Abhängig von Einsatzszenarien kann die Phrasensuche entweder sehr gute oder sehr schlechte Ergebnismenge liefern. Wenn man nach einen Title oder eine exakte Phrase in einem Dokument sucht, dann bekommt man gute Ergebnisse. Allerdings werden viele potenziell relevante Dokumente ausgeschlossen, weil die gesuchte Phrase in diesen Dokumenten nicht in der gleichen Wortfolge vorliegt. Sucht man beispielsweise nach *„relationale Datenbanken*", werden Dokumente mit *„relationale Datenbank*" nicht gefunden.

Fuzzy-Suche sorgt für Fehlertoleranz und erlaubt eine vordefinierte Abweichung zwischen den gesuchten und in Dokument vorhandenen Term. Das bekannteste Algorithmus für die Fuzzy-Suche ist Levenshtein-Distanz[15]. Das Algorithmus berechnet die Anzahl von Änderungen, die nötig sind, um ein Wort in ein anderes umzuwandeln. Je wenig Änderungen notwendig sind, desto ähnlicher sind die Worte.

Filter werden verwendet, um die Suchergebnisse nach bestimmten Kriterien einzuschränken. Der meistverbreitete Filter ist Seiten-*Pagination*. Die andere Einsatzmöglichkeiten sind Zeit- oder Typ-Filter.

2.4.6 Gewichtete Suche

In dem Kapitel *„Indexierung von Textdokumenten*" wurde bereits erwähnt, dass jeder Term von IR-System während der Indexierung gewichtet wird. Allerdings werden nicht nur Terme sondern auch die Suchanfragen(eng. search query) von IR-System gewichtet. Wenn die Suchanfrage aus mehreren einzelnen Suchanfragen besteht, dann wird die Gewichtung für alle einzelnen Anfragen ebenfalls berechnet.

Während die Volltextsuche die bessere Ergebnismenge durch Manipulation von Termen erreichen will, mischt sich die gewichtete Suche direkt in der Berechnung der Gewichtung ein. Dieses Verfahren ist als *Boosting* bekannt und wird meistens zur Anzeige von beliebtesten oder populärsten Dokumenten verwendet.

Es gibt drei unterschiedliche *Boosting*-Verfahren: ***Field-Boosting***, ***Term-Boosting*** und ***Query-Boosting***. ***Field-Boosting*** wird eingesetzt, wenn die Suche mehrere Felder anfragt und ein bestimmtes Feld höher als anderen priorisiert werden soll. Beim ***Term-Boosting*** wird die Suchanfrage durch Vorkommen eines bestimmten Terms höher gewichtet. Mit der ***Query-Boosting***, wie der Name schon sagt, kann eine *Query* geboosted werden.

[15] **https://de.wikipedia.org/wiki/Levenshtein-Distanz**.

2.4.7 Autovervollständigung

Autovervollständigung ist heutzutage fast in jeder Anwendung vorhanden. Die Möglichkeit, durch die Eingabe von wenigen Zeichen auf vermutlich relevanter Suchbegriff zu kommen, hat sich in der Realität als sehr praktisch erwiesen. Grundsätzlich basiert dieses Verfahren auf die Ergänzung von eingegebenen Anfangsbuchstaben. Trotzdem gibt es immer mehr Anwendungen, wo die eingegebene Buchstaben nicht als Anfangsbuchstaben sondern als einen Suchbegriff interpretiert werden um vermutlich relevante Dokumente sofort zur Auswahl anzuzeigen.

Mit Hilfe von Autovervollständigung wird das Erinnerungsproblem zu einem Wiedererkennungsproblem. Denn es reicht wenige Informationen, um ein gewünschter Suchbegriff zu treffen. Außerdem können auch Tippfehler bei der Autovervollständigung berücksichtigt und vermieden werden.

2.5 ElasticSearch

ElasticSearch ist ein Open-Source Suchserver, welcher auf Apache Lucene Bibliothek basiert. Das Projekt wird in der Programmiersprache Java geschrieben und ermöglicht die Kommunikation über REST-Schnittstelle. Die Daten werden in JSON-Format übertragen und in einem invertierten Index abgespeichert. Ein Index entspricht einer Datenbank. Jeder Datensatz wird durch Dokumenten und Spalten durch Felder repräsentiert. ElasticSearch verfolgt eine dokumentenorientierte Architektur und ist deswegen in der Lage auch komplexe Objekte in einem Feld zu speichern.

Relationale Datenbank	**Elasticsearch**
Datenbank	Index
Tabelle	Dokumenttype
Datensatz	Dokument
Spalte	Feld

Tabelle 2.1: Begriffsvergleich zwischen Relationale Datenbanken und ElasticSearch

ElasticSearch unterstützt alle im Kapitel 2.4 vorgestellte Methoden der Informationsbeschaffung und erweitert sie mit neuer Funktionalität. Zum Beispiel ist es möglich die Dokumente in Echtzeit zu indexieren. Die Indexierung wird dabei im Arbeitsspeicher gehalten und erst später auf die Festplatte geschrieben. Besonders mächtig ist ElasticSearch bei der Berechnung von Gewichtung(eng. score). Denn es werden mehrere Funktionen zur Manipulation von Score bereitgestellt. Es ist sogar erlaubt eigene Funktion zur Berechnung von Score zu schreiben. Standardmäßig wird aber Score nach Lucene-Formel berechnet[16].

[16] **www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html**.

$$score(q,d) = queryNorm(q) * coord(q,d) * \sum_{t(in)q} (tf * idf^2 * boost(t) * norm(t,d))$$

- $score(q,d)$ - Relevanz eines Dokumentes (d) für die Suchanfrage (q)
- tf - term frequency (Kapitel 2.4.2) - $tf = \sqrt{termFreq}$
- idf - inverse document frequency (Kapitel 2.4.2) - $idf = 1 + log(\frac{numDocs}{docFreq+1})$
- $queryNorm(q)$ - Funktion zur Berechnung der Normalisierung von *Search-Query* - $queryNorm(q) = 1/\sqrt{sumOfSquaredWeights}$, $sumOfSquaredWeights$ summiert IDF von allen Termen in der Suchanfrage
- $coord(q,d)$ - wird verwendet, um Dokumente mit mehrere vorhandenen Suchtermen höher zu gewichten $coord(q,d) = \frac{t(in)q}{t(in)d}$
- $boost(t)$ - erlaubt *boosting* von *Search-Query* (Kapitel 2.4.6). Standardmäßig ist die Wert gleich 1.
- $norm(t,d)$ - Funktion zur Berechnung von Feldnormalisierung $norm(t,d) = 1/\sqrt{numTerms}$, $numTerms$ steht für die Anzahl von Termen im gesuchten Feld

Eine wichtige Funktion, die in diesem Zusammenhang auch erwähnt werden soll, ist die *Decay*-Funktion. *Decay*-Funktion berechnet die Relevanz eines Dokumentes abhängig von der benutzerdefinierten Distanz zwischen einem nummerischen Feldwert und dem Original-Wert. Das nummerische Feld kann eine Zahl, ein Datum oder Geo-Location sein. Die Distanz wird *linear*, *exponential* oder standardmäßig nach *Gaus*-Algorithmus berechnet. Mit der folgenden Formel wird Distanz *exponential* berechnet.

$$S(doc) = exp(\lambda * max(0, |fieldValue - origin| - offset))$$

- $fieldValue$ - Feldwert
- $origin$ - Startwert, der bei der Distanzberechnung berücksichtigt wird. Beim Datum kann es zum Beispiel das heutiges Datum sein.
- $offset$ - Decau-Funktion wird berechnet, wenn die Distanz größer als $offset$. Standartwert ist 0.
- λ -sorgt dafür, dass die berechnete Distanz in einem benutzerdefinierten Intervall liegt. $\lambda = \frac{log(decay)}{scale}$, $decay$ definiert ein Abstand zwischen den Dokumenten. Standartwert ist 0.5. $scale$ ist die benutzerdefinierte Distanz.

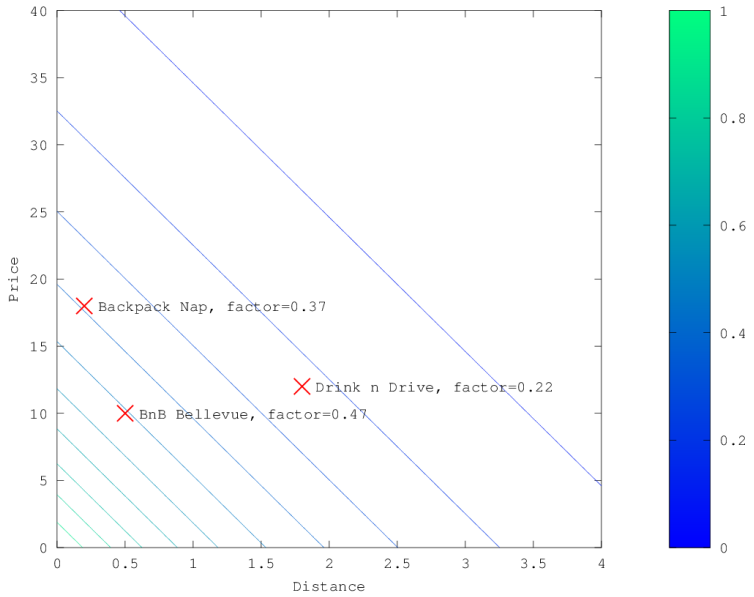

Abbildung 2.8: Visualisierung von *exponentialer Decay*-Funktion

2.5.1 Sharding

Das Wort *Elastic* in dem Projektname bezieht sich auf horizontale Skalierbarkeit, welche ermöglicht, die Daten auf mehrere Server zu verteilen. Alle Daten werden in einem Cluster vorgehalten. Ein Cluster besteht aus einem oder mehreren Knoten (eng. Node), die auf unterschiedlichen Server laufen können. Die Synchronisierung zwischen den Knoten wird über Cluster gesteuert und kann relativ einfach und flexibel von Benutzer konfiguriert werden. Wenn ein Server besonders gut aufgerüstet ist, ist es sinnvoll ein *Master*- und ein *Slave*-Node festzulegen.

Um Ausfallsicherheit zu gewährleisten, werden die Dokumente der einzelnen Indizes in mehrere *Shards* repliziert und auf den Knoten verteilt. Der *Shard*, welcher das Dokument zuerst speichert, wird *primary shard* genannt. Alle weitere *shards* sind *replica shards* und behalten nur die Kopie von diesem Dokument. Wenn ein *Node* ausfällt, kann das Dokument aus dem *replica shard* eines anderen Nodes gelesen werden[Rafal Kuc 2013]. Die Abbildung 2.9 stellt ein Cluster mit zwei Knoten, vier *primary shard* und ein *replica shard* dar.

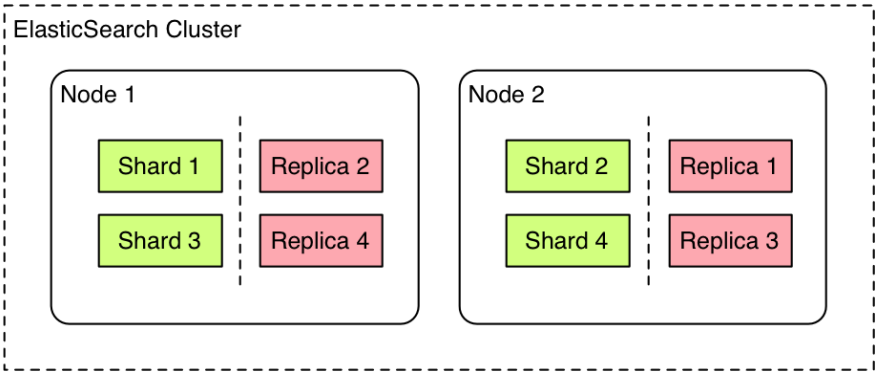

Abbildung 2.9: Beispiel - Ein Cluster mit zwei Knoten, 4 *shards* und 1 *replica shard*[Terrier 2013]

3. Analyse und Systementwurf

Dieses Kapitel legt fest, wie die neue Anwendungsanforderungen in der Implementierungsphase realisiert werden müssen. Dabei werden neue Prozesse detailiert beschrieben und mit Diagrammen visualisiert. Außerdem wird auch darüber diskutiert, welche Daten für die gestellten Anforderungen nötig sind und wie sie gespeichert werden müssen. Da die Anwendung *„IntelliJob"* bereits schichtenorientiert ist, müssen auch die neue Programmteile sinnvoll auf mehrere Schichten verteilt werden. Zuerst wird aber kurz angedeutet, warum *OpenNLP Framework*, das bereits in *„IntelliJob"* angewendet wird, zur Lösung von neuen Anforderungen nicht eingesetzt wird.

3.1 OpenNLP oder ElasticSearch

Mit Hilfe von *OpenNLP Framework* wurden bereits die Firmenadressen und die Kontaktpersonen aus den Arbeitsangeboten extrahiert und gespeichert. Deswegen war auch die erste Überlegung analog zu der alten Logik, die Arbeitsanforderungen ebenfalls mit *OpenNLP Framework* zu extrahieren und die extrahierte Daten mit *ElasticSearch* zu indexieren.

Die Muster-Erkennung von OpenNLP basiert auf künstlich erzeugten Trainingsdaten, die durch ein Lernprozess in einem Model gespeichert werden. Die Effektivität des Modells hängt im größten Teil von der Anzahl und Qualität der bereitgestellten Trainingsdaten ab. Der Hersteller verspricht gute Ergebnisse mit mindestens 15.000 Datensätzen. Die weitere Möglichkeit das Modell zu verbessern sind sogenannten *Features*. Das sind besondere Text-Merkmale, die versuchen, in den Datensatz enthaltenen Schlüsselinformationen zu extrahieren[Brad 2015].

Die Erstellung des Modells ist also ein sehr aufwendiger und komplexer Prozess und garantiert auch nicht, dass alle Arbeitsanforderungen aus den Arbeitsangeboten erfolgreich extrahiert werden. Die weitere Überlegungen haben zu der Frage geführt: *Wieso müssen die Arbeitsanforderungen überhaupt erstmal extrahiert und erst dann mit ElasticSearch indexiert werden?* Stattdessen können alle Arbeitsangebote mit *ElastSearch* indexiert werden. Danach können auch die Benutzerfähigkeiten ebenfalls indexiert werden und die Suche kann in dem Fall auch über Dokumenten erfolgen. Da das komplette Arbeitsangebot und nicht nur ein Teil davon mit Benutzerfähigkeiten durchgesucht und verglichen wird, kann die Relevanz eines Arbeitsangebotes viel genauer bestimmt werden.

Bei der Integrierung von *ElasticSearch* in die bereits vorhandene Infrastruktur muss darauf geachtet werden, dass die Anwendung *„IntelliJob"* eine *Standalone* Web-Anwendung(siehe Kapitel 2.2.1) ist. Deswegen muss auch *ElasticSearch* mit der JAR-Datei ausgeliefert und mit der Anwendung zusammen gestartet werden. Da die Anwendung lokal auf einer Maschine ausgeführt wird, besteht auch keine Möglichkeit die Ausfallsicherheit und Lastverteilung(siehe Kapitel 2.5.1) zu realisieren. Deswegen müssen alle *ElasticSearch* Indexe nur auf einem *Node* und einem *Shard* gespeichert werden.

3.2 Prozessablauf

Grundsätzlich gibt es nur zwei Prozesse, die neue implementiert werden müssen.

Der erste Prozess ist die Erfassung von Benutzerdaten. Dafür muss eine neue Webform entwickelt werden, die es ermöglicht, die Benutzerdaten aufzunehmen. Da *„IntelliJob"* immer lokal auf einem Rechner ausgeführt wird, ist es legitim, dass nur ein Benutzer auf das System gleichzeitig zugreifen darf. Der Mehrbenutzerzugriff wird in dieser Version nicht unterstützt. Wenn die Anwendung zum ersten Mal gestartet wird, ist das Benutzerprofil leer und kann sofort angepasst werden. Der Benutzer hat ein Auswahl zwischen einem einfachen und erweiterten Suchfilter. Bei dem erweiterten Suchfilter muss der Benutzer die ausgewählte *Skills* und Fähigkeiten mit einer Skala von 1 sehr schlecht bis 5 sehr gut bewerten.

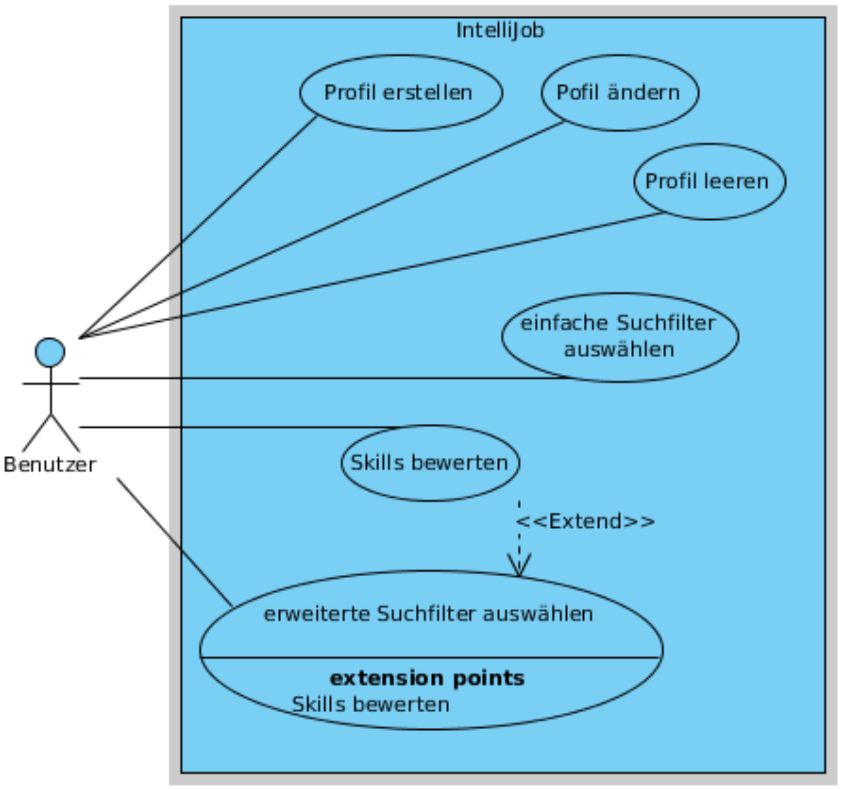

Abbildung 3.1: Benutzerprofil - Use Case Diagramm

Alle Benutzerdaten müssen in *MongoDB* gespeichert werden. Die Daten, dir für die Suche relevant sind, müssen noch zusätzlich in *ElasticSearch* gespeichert werden. Der Vorteil von dieser Trennung liegt darin, dass die Suchdaten nur einmalig beim Speichern aggregiert und danach immer wieder benutzt werden können. Die Abbildung(3.2) stellt der Prozess als Sequenz Diagramm dar.

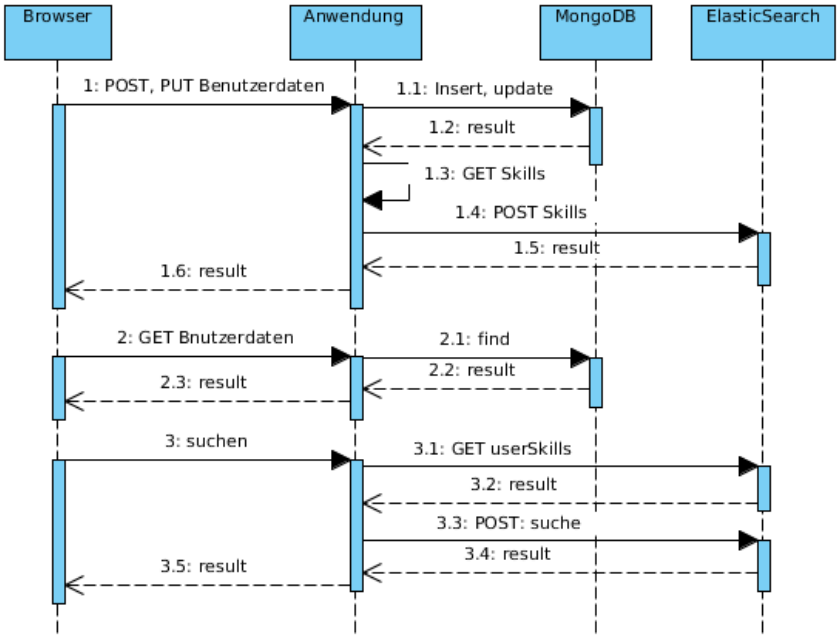

Abbildung 3.2: Benutzerprofil - Sequence Diagram

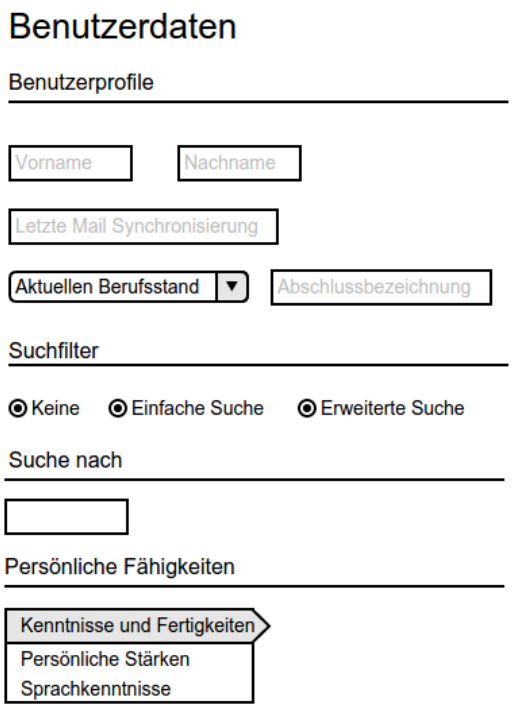

Abbildung 3.3: Benutzerprofil - MockUp

Der zweite Prozess ist die Indexierung von Arbeitsangeboten. Alle Arbeitsangebote müssen mit einem deutschen *Analyzer* indexiert werden(siehe Kapitel 2.4.2). Bevor ein Arbeitsangebot gespeichert wird, muss es geprüft werden, ob das Arbeitsangebot bereits in Datenbank existiert. Dafür muss eine Hash-Summe von Plain-Text gebildet und in Datenbank gespeichert werden. Dadurch wird es verhindert, dass die gleiche Arbeitsangebote doppelt gespeichert werden(siehe Kapitel 2.2.3). Wenn ein Arbeitsangebot bereits existiert, muss es durch ein neues Arbeitsangebot ersetzt werden.

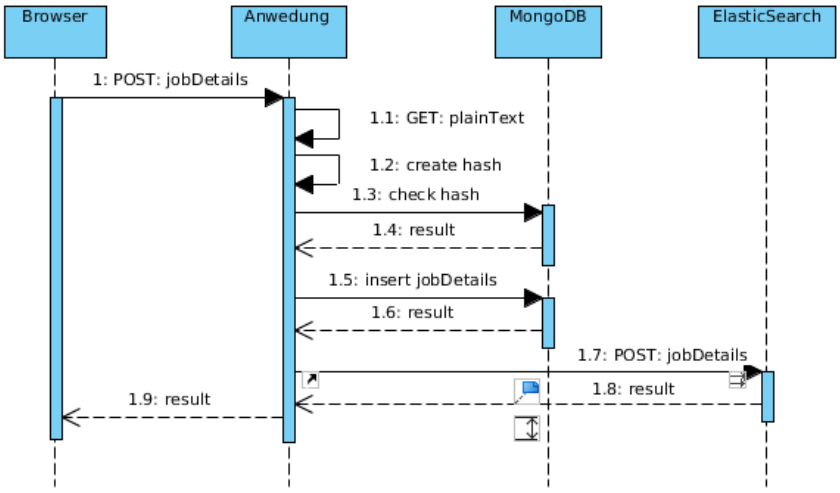

Abbildung 3.4: Arbeitsangebote - Sequence Diagram

Nachdem die Benutzerdaten erfasst und die Arbeitsangebote erfolgreich indexiert wurden, kann auch die Suche implementiert werden. Bei der Suche müssen die Benutzerdaten, wie ausgewähltes Suchfilter und die eingegebene Suchdaten, berücksichtigt werden. Außerdem muss der Benutzer in der Lage sein, der Suchfilter jede Zeit ändern zu können, ohne sein Profil dabei editieren zu müssen. Falls erweiterter Suchfilter ausgewählt wird, müssen neben der persönlichen Fähigkeiten auch ihre Bewertungen berücksichtigt werden.

Um die alte und bereits gelesene Arbeitsangebote aus dem Suchvorgang auszuschließen, muss es ermöglicht werden, die Arbeitsangebote als gelesen zu markieren. Falls ein Arbeitsangebot existiert und überschrieben werden soll, darf die Markierung (gelesen) nur dann entfernt werden, wenn das Arbeitsangebot älter als ein Monat ist.

Außerdem muss es für den Benutzer ersichtlich werden, nach welchen Kriterien gerade gesucht wird. Die Repräsentation von erweiterten Suchdaten muss mit *Tag clouds*(siehe Kapitel 2.3) realisiert werden.

Die einzelne nicht-funktionale Systemanforderung ist möglichst höhe Abstraktion der *ElasticSearch-Query* von restlichen Programmkomponenten, damit die *Query* sinnvoll evaluiert werden könnte.

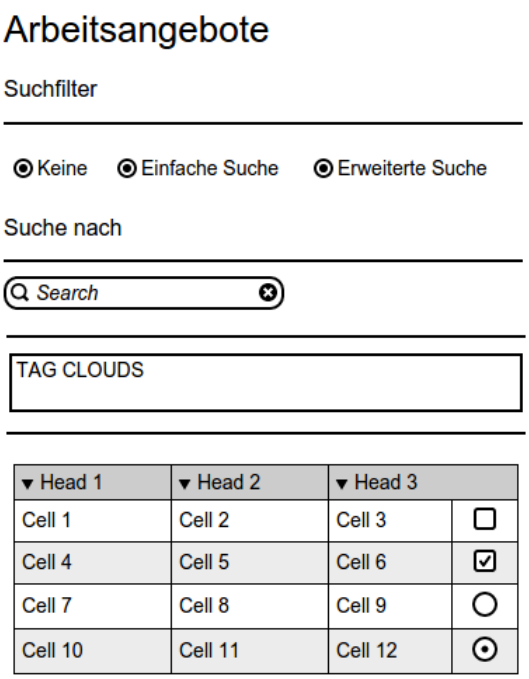

Abbildung 3.5: Suchmaske-MockUp

3.3 Datenstruktur

Am besten lässt sich die Datenstruktur von dokumentenorientierten Datenbanken mit Klassendiagrammen abbilden. Da die Daten sowohl in *MongoDB* als auch in *ElasticSearch* gespeichert werden, soll auch unterschiedliche Datenstrukturen bereitgestellt werden.

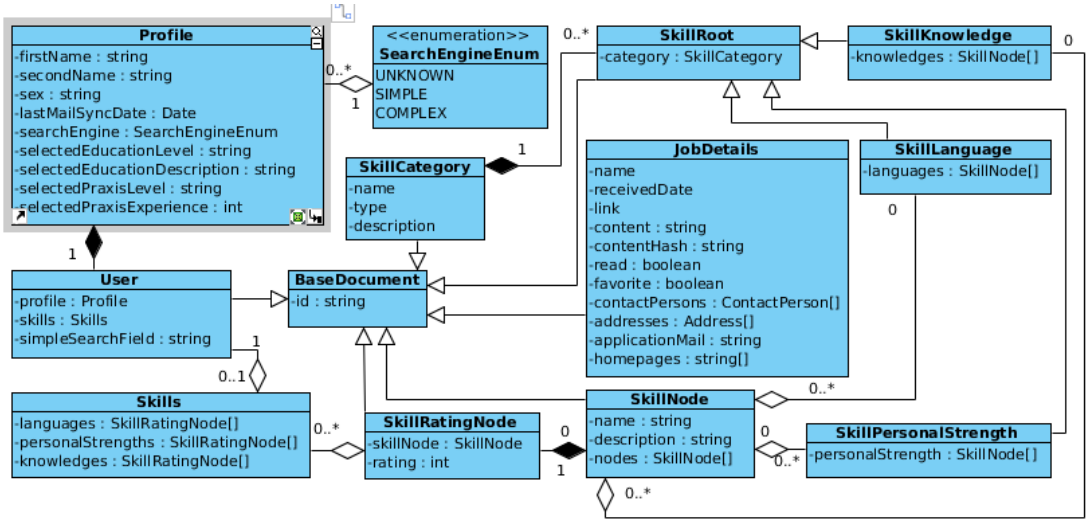

Abbildung 3.6: MongoDB-Datenstruktur

Alle private Attribute im Klassendiagramm beinhalten auch die GETTER- und SETTER-Methoden, die aber nicht explizit eingegeben werden.

Die Klasse *JobDetails* wird um zwei neue Attribute erweitert. Die Attribute *contentHash* speichert den Hash-Wert von Arbeitsangebot ab und die boolesche Attribute *read* steht für die Markierung *gelesen* oder *nicht gelesen*. Standardmäßig muss das Feld *read* gleich *false* sein und erst durch Benutzeraktion überschrieben werden. Das Feld *content* beinhaltet die vollständige Beschreibung des Arbeitsangebotes in der HTML-Form.

Alle von Systemunterstützte persönliche Fähigkeiten und Qualifikationen werden als Baumstruktur[1] abgebildet und dem Benutzer auch in dieser Form repräsentiert. Die Klasse *SkillRoot* ist das erste Knote des Baumstruktur und beschreibt auch die zugehörige Kategorie, die durch Klasse *SkillCategory* repräsentiert wird. In dieser Version werden nur drei verschieden Kategorie unterstützt: *Kenntnisse und Fertigkeiten*, *Persönliche Stärken* und *Sprachen*. Jede Kategorie besitzt eigene Klasse (*SkillKnowledge*,*SkillsPersonalStrength*,*SkillLanguage*), die von der Klasse *SkillRoot* abgeleitet wird. Jedes *Root*-Element eines Baums kann beliebig viele Knoten(*SkillNode*) besitzen und jede Knote kann wiederum beliebig viele Unterknoten haben.

Die Klasse *User* fasst die Benutzerdaten zusammen. Die *Enum*-Attribute *searchEngine* in der Klasse *Profile* bezieht sich an den ausgewählten Suchverfahren, wo *SIMPLE* für die einfache und *COMPLEX* für die erweiterte Suche steht. Die Daten, die für die Suche relevant sind, werden in *User* Klasse ausgelagert. So beinhaltet das Feld *simpleSearchField* die von Benutzer eingegebene Suchdaten, die bei der einfachen Suche gelesen werden müssen. Die Klasse *Skills* enthält alle ausgewählte persönliche Fähigkeiten und Qualifikationen mit ihrer Bewertungen. Grundsätzlich können nur die letzten Knoten des Baumstruktur ausgewählt und von Benutzer bewertet werden. Diese Daten werden in der Klasse *SkillRatingNode* zusammengefasst und genau diese Daten müssen auch bei der erweiterten Suche berücksichtigt werden.

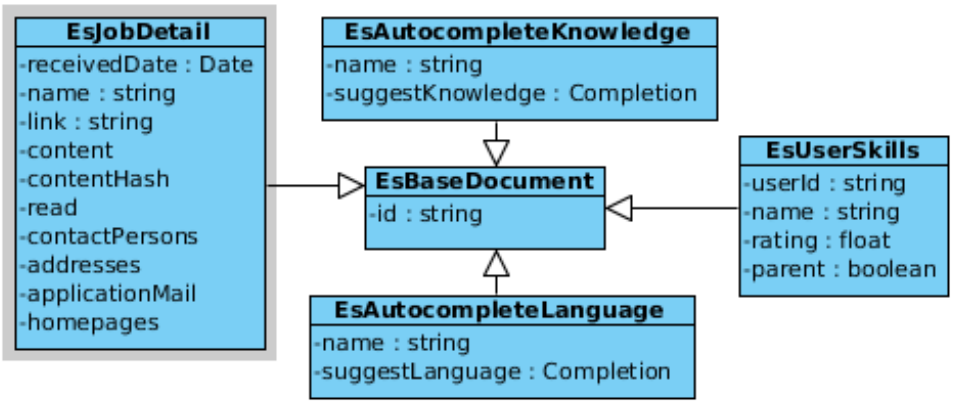

Abbildung 3.7: ElasticSearch-Datenstruktur

[1] **https://de.wikipedia.org/wiki/Baum_(Graphentheorie)**.

Die *ElasticSearch*-Klasse *EsJobDetail* ist äquivalent zu der Klasse *JobDetails* in *Mongo* Datenbank. Der einzige wesentliche Unterschied liegt im Feld *content*, das den Inhalt eines Arbeitsangebotes nicht als HTML sondern als Plain-Text speichert. Dieses Feld muss mit einem deutschen *Analyzer* indexiert und für die Suche optimiert werden.

Die Klasse *EsUserSkills* muss nur die von dem Benutzer bewertete Qualifikationen und ihre übergeordnete Elemente indexieren. Da nicht die vollständige Baumstruktur sondern nur für die Suche relevante Dokumente indexiert werden, muss es eine Möglichkeit geben, die *Child*-Elementen von *Parent*-Elementen zu unterscheiden. Dafür muss die Klasse um ein neues boolesches Feld *parent* erweitert werden.

Die Klassen *EsAutocompleteKnowledge* und *EsAutocompleteLanguage* indexieren alle von System unterstützte persönliche Qualifikationen und Sprachen, um den Benutzer bei der Dateneingabe mit einer optionalen *Autocomplete*-Funktion unterstützen zu können.

Damit die Dokumente, die in beiden Systemen vorhanden sind, eindeutig identifiziert werden könnten, müssen ihre *IDs* auch gleich sein.

3.4 Schichtenarchitektur

In dem Kapitel 2.2.2 wurde die Schichtenarchitektur der Anwendung bereits beschrieben. Die neue Anforderungen müssen deswegen die alte Architektur beibehalten.

Domain: Jedes Dokument in Datenbank muss durch ein Domain-Objekt repräsentiert werden. Das Domain-Objekt beschreibt nur die Daten, die persistiert oder verarbeitet werden müssen und keine Information darüber enthält, wo und wie diese Daten liegen. Dank dieser Abstraktion wird die Datenbankschicht von der Anwendung getrennt gehalten. Der weitere Vorteil von diesem Einsatz besteht darin, dass alle Datenbankoperationen an einem Domain-Objekt ausgeführt werden und die Daten nicht extra umgewandelt werden müssen. Allerdings verliert man dadurch die Flexibilität der Datenstruktur, die einen dokumentenorientierten Einsatz bietet, weil die Datenstruktur durch Domain-Objekt festgelegt und zur Laufzeit nicht mehr geändert werden kann. Alle Domain-Objekten wurden bereits in dem vorherigen Kapitel definiert und müssen auch genau so in der Anwendung implementiert werden.

Datenzugriffsschicht: Der Zugriff auf Datenbank erfolgt über Repository. Ein Repository beschreibt alle Operationen, die an einem Domain-Objekt ausgeführt werden dürfen. Außerdem behält ein Repository eine Referenz auf das persistierte Objekt und kann es beim nächsten Zugriff nicht aus der Datenbank sondern aus der eigenen Repository laden[Vladimir Gitlevich 2006, S.53]. In der Anwendung müssen alle Repository-Klassen entweder von *MongoRepository* oder von *ElasticsearchRepository* abgeleitet werden. Die beide Repositories werden von *Spring Framework* bereitgestellt und bieten viele Datenzugriff-Methoden an.

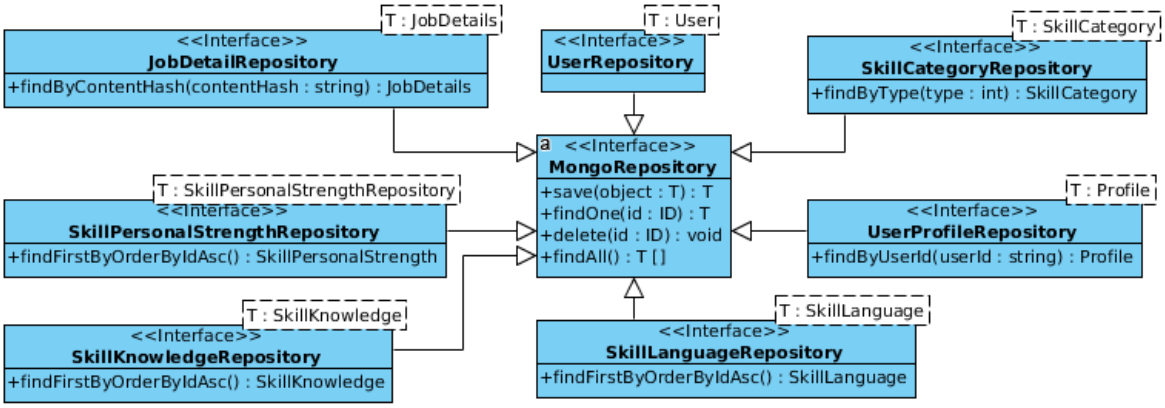

Abbildung 3.8: Mongo - Datenzugriffsschicht

Alle Methoden von *MongoRepository* sind generisch und müssen nicht implementiert werden. Um die Methoden von *MongoRepository* nutzen zu können, müssen alle abgeleitete Schnittstellen das *Template*-Objekt typisieren.

Die Methode *findByContentHash* in der Klasse *JobDetailRepository* muss ein Dokument, dessen *ContentHash* gleich dem übergebenen *ContentHash* finden und zurückgeben.

Die *Skill-Repositories* werden um eine neue Methode *findFirstByOrderByIdAsc* erweitert, die immer das erste Dokument zurückgibt.

Die Methode *findByUserId* muss das Profil eines Benutzers zurückliefern.

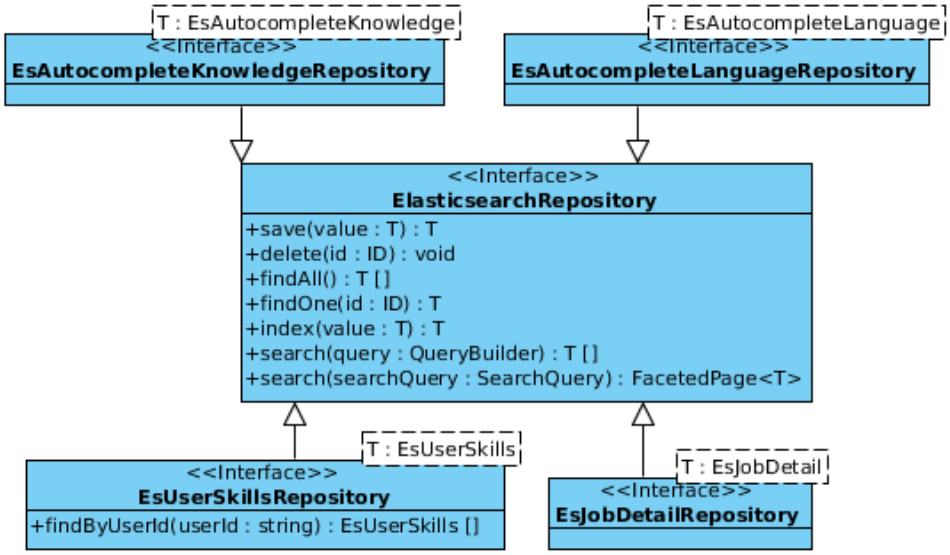

Abbildung 3.9: ElasticSearch - Datenzugriffsschicht

Die *ElasticSearchRepository* im Vergleich zur *MongoRepository* bietet noch zusätzliche Methoden zur Erstellung von Index- und Suchanfragen. Die einzige Methode, die noch extra implementiert wird, ist *findByUserId* und die muss alle indexierte Qualifikationen und Fähigkeiten eines Benutzers zurückgeben.

Controlschicht: vereint alle Methoden aus Datenzugriffsschicht und führt alle nötige Operationen aus, um die Benutzeranfrage zu bearbeiten. Das ist der Kern der Anwendung. Denn fast alle Entscheidungen werden in dieser Schicht getroffen.

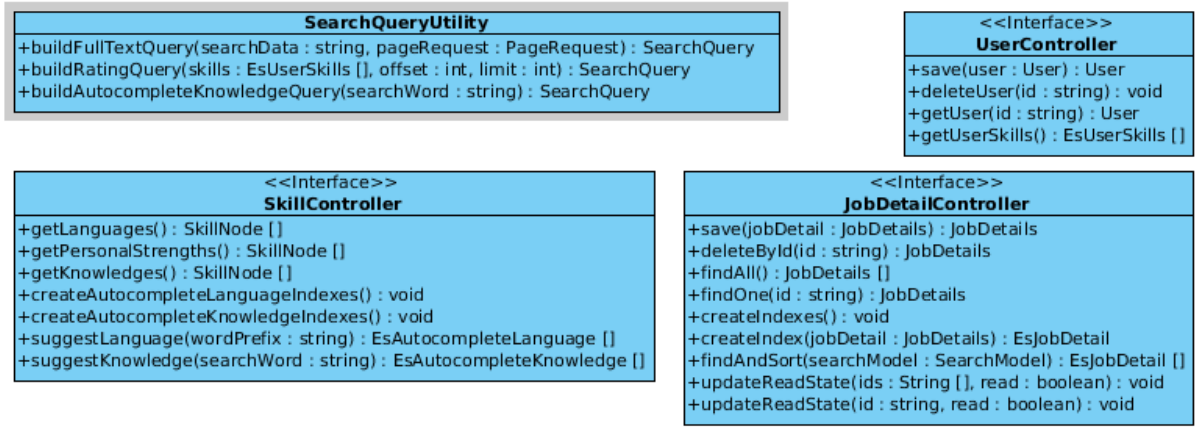

Abbildung 3.10: Controlschicht

Die Klasse *JobDetailController* vereint alle Operationen, die an einem Arbeitsangebot ausgeführt werden. Beim Speichern (*save*) muss das Arbeitsangebot in *MongoDB* geschrieben und indexiert werden. Um *ElasticSearch*-Indexes anzulegen, müssen die Methoden *createIndex* und *createIndexes* implementiert werden. Bei *createIndexes* müssen alle Arbeitsangebote indexiert werden. Wenn ein Arbeitsangebot gelöscht oder geändert wird, muss es sowohl in *MongoDB* als auch in *ElasticSearch* aktualisiert werden. Die Suche in Arbeitsangeboten wird mit der Methode *findAndSort* realisiert. Der Parameter *SearchModel* muss alle für die Suche relevante Daten beinhalten: Benutzer, Suchverfahren, Limit und *search value*. Die Entscheidung, welche Such-*Query* gebildet werden muss, hängt von dem übergebenen Suchverfahren ab.

Die Implementierung von Such-*Queries* wird in die Utility-Klasse *SearchQueryUtility* ausgelagert und kann über statische Methoden aufgerufen werden. Die Methode *buildFullTextQuery* muss eine einfache und die *buildRatingQuery* eine erweiterte Such-*Query* bauen. Die Methode *buildAutocompleteKnowledgeQuery* implementiert eine *Autocomplete*-Suche für allen vom System unterstützten Kenntnissen und Fertigkeiten.

Bevor die *Autocomplete*-Suche verwendet werden kann, müssen alle Kenntnissen und Sprachen indexiert werden. Das wird mit den Methoden *createAutocompleteLanguageIndexes* und *createAutocompleteKnowledgeIndexes* der Klasse *SkillController* realisiert.

Die Implementierungsklasse von *UserController* verwaltet alle Aktionen zur Manipulation von Benutzerdaten. Die Methode *save* muss nicht nur die Benutzerdaten in MongoDB speichern, sondern auch die ausgewählte und bewertete persönliche Fähigkeiten indexieren und mit der Methode *getUserSkills* zur Verfügung bereitstellen.

Serviceschicht: wird als RESTFul-Service implementiert. Beim RESTful handelt es sich um einem ressourcenorientierten Webservice, dessen Ressourcen über HTTP-Methoden angefragt werden.[Richardson u. a. 2008, S.13]. Die Kenntnisse über RESTful-Technologie werden vorausgesetzt und werden deswegen in dieser Arbeit nicht weiter beschrieben.

Ein Arbeitsangebot als gelesen bzw. als nicht gelesen markieren

`PUT: /api/jobdetails/:id`

Parameter

Name	Type	Beschreibung
read	boolean	**Required.** true für gelesen, false für nicht gelesen

Alle Arbeitsangebote als gelesen bzw. als nicht gelesen markieren

`PUT: /api/jobdetails`

Parameter

Name	Type	Beschreibung
ids	array	**Required.** Liste mit ids, die aktualisiert werden müssen
read	boolean	**Required.** true für gelesen, false für nicht gelesen

Response

Code	Beschreibung
202	*Read*-Status wurde erfolgreich aktualisiert.

Arbeitsangebot löschen

`DELETE: /api/jobdetails/:id`

Response

Code	Beschreibung
200	Arbeitsangebot wurde erfolgreich gelöscht. Das gelöschte Arbeitsangebot muss zurückgegeben werden.
404	Arbeitsangebot wurde nicht gefunden.

Suchen nach passenden Arbeitsangebote

`GET: /api/jobdetails/:offset/:limit`

Parameter

Name	Type	Beschreibung
searchFilter	string	**Optional.** Ausgewähltes Suchverfahren Erlaubt: UNKNOWN, SIMPLE, COMPLEX
searchData	string	**Optional.** Benutzerdefinierte Suchdaten für die einfache Suche

Response

Code	Beschreibung
200	Alle gefundene Arbeitsangebote werden zurückgegeben.

Neuer Benutzer anlegen

```
POST: /api/users
```

Payload-JSON

```
{
  "profile": {
    "firstName": "fisrstName",
    "secondName": "secondName",
    "lastMailSyncDate": "2015-10-07 20:36:26",
    "searchEngine": "COMPLEX",
    "selectedEducationLevel": "Hochschulabschluss - Bachelor",
    "selectedEducationDescription": "Bachelor Angewandte Informatik",
    "selectedPraxisLevel": "Student",
    "selectedPraxisExperience": 3
  },
  "simpleSearchField": "Datenbanken",
  "languages": [{
     "skillRatingData": {
       "skillData": {
         "id": "563537b444ae353683c50777",
         "name": "Russisch"
       },
       "rating": 5
     }
   }],
  "personalStrengths": [],
  "knowledges": []
}
```

Response

Code	Beschreibung
200	Der Benutzer wurde erfolgreich erstellt.

Benutzer löschen

```
DELETE: /api/users/:id
```

Response

Code	Beschreibung
200	Der Benutzer wurde erfolgreich gelöscht
404	Benutzer wurde nicht gefunden.

Benutzer ändern

```
PUT: /api/users/:id
```

Payload ist gleich wie beim Anlegen eines neuen Benutzers.
Response

Code	Beschreibung
200	Der Benutzer wurde erfolgreich geändert
404	Benutzer wurde nicht gefunden.

Arbeitsangebote indexieren

```
PUT: /api/elastic/jobdetails/indexes
```

Sprachen für Autocomplete indexieren

```
PUT: /api/elastic/autocomplete/language/indexes
```

Kenntnissen und Fertigkeiten für Autocomplete indexieren

```
PUT: /api/elastic/autocomplete/knowledge/indexes
```

Response ist überall gleich.

Code	Beschreibung
202	Daten wurden erfolgreich indexiert

Suchen nach Sprachen

```
GET: /api/elastic/autocomplete/language/name/:value
```

Response

Code	Beschreibung
200	Liefert alle gefundene Sprachen

Suchen nach Kenntnissen und Fertigkeiten

```
GET: /api/elastic/autocomplete/language/name/:value
```

Response

Code	Beschreibung
200	Liefert alle gefundene Kenntnissen und Fertigkeiten

Präsentationsschicht: ist eine eigenständige Webanwendung, die mit *AngularJS* entwickelt wird. In der moderne Webentwicklung werden viele Programmkomponente clientseitig ausgeführt. Deswegen kann auch das Frontend-Code sehr groß sein und muss gut strukturiert werden. Die neue Benutzerprofilseite muss ebenfalls mit *AngularJS* implementiert und in drei neuen Dateien verteilt werden:

- *user.html*- statische HTML-Code
- *userControllers.js* - steuert Benutzeraktion und sorgt für dynamischen Content der Benutzerprofilseite.
- *userServices.js* - enthält Funktionen, um REST-Services asynchron aufzurufen.

Die Repräsentation von Arbeitsangeboten muss einfach mit der Such-Form erweitert werden.

4. Entwicklung und Implementierung

In diesem Kapitel wird die Implementierungsphase beschrieben. Da viele funktionale Anforderungen und Recherchen in früheren Kapitel detailiert beschrieben wurden, werden hier nur die wichtigste Programmteile dokumentiert. Das komplette Programmcode kann entweder von GitHub oder angehängte CD geladen werden.

Die Implementierung beginnt mit der Konfiguration von *ElasticSearch*. Danach werden die Daten indexiert und gespeichert. In dem Kapitel Indexierung von Arbeitsangeboten wird auch die Lösung für duplizierten Arbeitsangeboten vorgestellt. Wenn *ElasticSearch* erfolgreich konfiguriert und alle Daten richtig indexiert sind, kann auch die Suche implementiert werden. Anschließend werden einige Front-End Komponente vorgestellt.

4.1 ElasticSearch - Konfiguration

Der Hauptvorteil von Spring Boot Technologie liegt darin, dass die komplette Spring-Web-Infrastruktur automatisch initialisiert und konfiguriert wird. Damit stellt Spring Boot eine vollständige Laufzeitumgebung zur Verfügung. Um *ElasticSearch* in das Projekt anzubinden, muss der Entwickler einfach die Konfiguration beschreiben.

Als erstes muss eine neue *Maven-Dependency* hinzugefügt werden.

```
<dependency>
   <groupId>org.springframework.data</groupId>
   <artifactId>spring-data-elasticsearch</artifactId>
   <version>1.2.2.RELEASE</version>
</dependency>
```

Alle *Config*-Daten werden als Key-Value Paar in einer *Property*-Datei abgelegt.

```
# EMBEDDED ELASTICSEARCH
elasticsearch.host=127.0.0.1
elasticsearch.path.data=data
elasticsearch.clusterName=civis
elasticsearch.http.enabled=true
elasticsearch.http.cors.enabled=true
```

Auf diese Properties können alle Konfigurationsklassen mittels Annotation *@Value*

zugreifen.

```
/**
 * ElasticSearch Configuration class.
 */
@Configuration
@EnableElasticsearchRepositories(basePackages =
    "com.intellijob.elasticsearch.repository")
public class ElasticsearchConfiguration {

   @Value("${elasticsearch.host}")
   private String host;

  ...
}
```

Letztendlich muss noch *ElasticSearch*-Client konfiguriert werden.

```
...
   /**
    * Init elasticsearch client and open connection to cluster.
    *
    * @return default ElasticsearchTemplate connected to cluster.
    */
   @Bean
   public ElasticsearchTemplate elasticsearchTemplate() {
      LOG.info("Start elasticsearch server");
      NodeBuilder nodeBuilder = new NodeBuilder();
      nodeBuilder.clusterName(clusterName).local(false);
      if (StringUtils.hasLength(pathData)) {
         nodeBuilder.settings().put("path.data", pathData);
      }
      nodeBuilder.settings().put("http.enabled", httpEnabled);
      nodeBuilder.settings().put("http.cors.enabled", httpCorsEnabled);
      nodeBuilder.settings().put("network.host", host);

      node = nodeBuilder.node();
      return new ElasticsearchTemplate(node.client());
   }
...
```

Das ist schon alles, was gemacht werden muss. Jetzt kann *ElasticSearch* von *Spring Boot* genau wie beschrieben konfiguriert werden. Es wird *local* gestartet und alle Daten werden unter *{project}/data* Verzeichnis abgelegt. Der Cluster *civis* hat nur ein *Node* und ist über HTTP erreichbar. Dank der *Property elasticsearch.http.cors.enabled* können auch andere Anwendungen auf *ElasticSearch* zugreifen. Das ist nötig, wenn *ElasticSearch* zusammen mit anderen *Plugins* verwendet werden soll. Die vollständige Konfigurationsklasse wird angehängt.

4.2 Daten Indexierung

Da die Datenstruktur in Kapitel 3.3 ausführlich beschrieben wurde und nur noch in Java-Programmcode umgewandelt werden soll, wird in diesem Kapitel nur auf Indexierung von Daten mit *ElasticSearch* eingegangen.

4.2.1 Arbeitsangeboten

Jede Domain-Klasse wird als Dokument annotiert. Die Annotation erwartet als Parameter der Indexname. Die weitere Parameter, wie *type*, *shards* und *replicas*, sind optional. Die Standartwerte für *shards* und *replicas* sind 5 und 1. Deswegen müssen sie entsprechend im Kapitel 3.1 definierten Systemanforderungen überschrieben werden.

```
/**
 * Elasticsearch document representation of domain object
    <code>JobDetail</code>.
 */
@Document(indexName = "intellijob", type = "jobDetails", shards = 1,
    replicas = 0)
public class EsJobDetail extends EsBaseDocument {
...
```

Da die *shard* und *replicas* Einstellungen bei dem Dokument definiert werden, ist es möglich unterschiedliche Dokumente je nach Wichtigkeit auch unterschiedlich zu replizieren.

Die Feldindexierung wird auch mittels Annotation (*@Field*) definiert. Die Einstellung *index = FieldIndex.analyzed* sorgt für die Feldindexierung. Standardmäßig werden alle Felder indexiert. Wenn ein Feld nicht indexiert werden soll, dann muss es mit *FieldIndex.not_analyzed* überschrieben werden. Wenn *Analyzer* für ein String-Feld nicht gesetzt wird, werden alle Worte ohne Umformung einfach nach Leerzeichen gesammelt und indexiert.

```
...
@Field(
  type = FieldType.String,
  index = FieldIndex.analyzed,
  searchAnalyzer = "german",
  indexAnalyzer = "german",
  store = true
)
private String content;
...
```

Das Arbeitsangebot wird beim Speichern zur Laufzeit indexiert. Leider ist es in *ElasticSearch* nicht möglich Indexe zu überschreiben. Deswegen müssen die Indexe

vor der Indexierung erstmal gelöscht werden.

```
/**
 * This is an implementation of interface <code>JobDetailController</code>.
 */
@Controller
public class JobDetailControllerImpl implements JobDetailController {
...
   /**
    * {@inheritDoc}
    */
   @Override
   public JobDetail save(JobDetail jobDetail) {
      //save in mongo
      jobDetailRepository.save(jobDetail);

      if (esJobDetailRepository.exists(jobDetail.getId())) {
         // remove elasticsearch index if exists
         esJobDetailRepository.delete(jobDetail.getId());
      }

      // map to elasticSearch object; job content is a plain text!
      EsJobDetail esJobDetail = mapTo(jobDetail);

      //create elasticsearch index
      esJobDetailRepository.index(esJobDetail);

      return jobDetail;
   }
...
```

Die Methode *mapTo* initialisiert das Objekt *EsJobDetail* mit allen vorhandenen Arbeitsangebotsdaten und ersetzt das HTML-Content durch Plain-Text.

Bevor die Methode *save* aufgerufen wird, muss es auch geprüft werden, ob das Arbeitsangebot mit dem gleichen Inhalt schon gespeichert wurde. Deswegen wird die Methode *save* nicht direkt sondern innerhalb des Datenextraktion ausgeführt. Dieser Prozess wird durch Benutzeraktion gestartet.

Nachdem die Daten erfolgreich extrahiert wurden, wird ein Hashwert mit der Hash -Funktion *SHA-512* von *Plain*-Text gebildet und es wird versucht ein Dokument mit diesem Hashwert in Datenbank zu finden. Wenn ein Dokument gefunden wird, werden die alte Daten erst dann überschrieben, wenn die neue Daten nicht *null* und nicht leer sind. Es wurde schon erwähnt, dass einige Daten mit *OpenNLP Framework* mittels trainierten Modell extrahiert werden. Dadurch kann unter Umständen(z.B. Fehler in dem Modell) passieren, dass das neue Modell weniger Daten findet. Mit dem oben beschriebenen Einsatz kann es verhindert werden, dass die alte gefundene Daten durch neue leere Daten überschrieben werden. Gemäß der Anforderungen wird es natürlich auch geprüft, ob das Flag-*read* überschrieben

werden soll.

```
...
if (!isTimeToOverwrite(newJobDetail.getReceivedDate(),
    oldJobDetail.getReceivedDate())) {
  // read state should not be overwritten.
  newJobDetail.setRead(oldJobDetail.isRead());
}
...
```

Die neue Erweiterung der Datenextraktion und Indexierung sieht wie folgt aus.

```
...
   /**
    * {@inheritDoc}
    */
   @Override
   public JobDetail extractJobDetailAndSave(Job job) {
      JobDetail jobDetail = extractJobDetail(job);

      JobDetail similarityJob = jobDetailRepository
            .findByContentHash(jobDetail.getContentHash());
      if (similarityJob == null) {
         JobDetail persistedJobDetail = save(jobDetail);
         jobController.setExtractedFlag(job, Boolean.TRUE);
         return persistedJobDetail;
      } else {
         JobDetail mergedJobDetail = mergeJobDetails(jobDetail,
             similarityJob);
         save(mergedJobDetail);
         jobController.setExtractedFlag(job, Boolean.TRUE);
         return mergedJobDetail;
      }
   }
...
```

Alle ausgewählte und bewerte persönliche Fähigkeiten und Kenntnisse werden analog zu Arbeitsangeboten indexiert. Außerdem werden dabei auch die übergeordnete Elemente automatisch bewertet. Das Ranking des übergeordnetes Elementes wird durch die Anzahl von ausgewählten Unterelementen bestimmt und gespeichert.

4.2.2 Autovervollständigung

Interessante Erweiterung gibt es für die Felder, die bei der Autovervollständigung benutzt werden. Die Autovervollständigung wird mit der *Suggest-Query*[1] realisiert.

[1] **www.elastic.co/guide/en/elasticsearch/reference/current/search-suggesters-completion.html**.

Wenn nach der Sprache gesucht wird, werden alle Sprachen, die äquivalent zu den eingegebenen Anfangsbuchstaben sind, als einfache Worte zurückgesendet. Es wird also kein Objekt sondern wirklich nur ein Wort-Term gesendet. Um das Wort mit dem Objekt identifizieren zu können, sollen alle nötige Daten als *Payload* zu diesem Wort mitgesendet werden. Die Annotation *@CompletionField(payloads = true)* legt fest, welches Feld bei der *Suggest-Query* gelesen werden muss und ob das *Payload* mitgesendet wird.

```
@Document(indexName = "autocomplete", type = "languages", shards = 1,
    replicas = 0)
public class EsAutocompleteLanguage extends EsBaseAutocomplete {

/**
* Thi field is required to create an index for autocomplete service.
*/
@CompletionField(payloads = true)
protected Completion suggestLanguage;
...
```

Das *Payload* wird bei der Initialisierung eines Objektes mit Hilfe von Konstruktor erstellt und beinhaltet nur die *Id* und der Name des Dokumentes. Das Objekt *Completion* erwartet ein Array mit Daten, die für dieses Dokument relevant sind und indexiert werden müssen.

```
   public EsAutocompleteLanguage(String id, String name, Boolean
       withSuggest) {
      super(id);
      setName(name);
      if (withSuggest) {
         this.suggestLanguage = new Completion(name.split(" "));
         Map<String, Object> payload = createPayload();
         this.suggestLanguage.setPayload(payload);
      }
   }
```

4.3 Suche

Die Suche ist eine zentrale Thema dieser Arbeit. Es werden mehrere Such-*Query* schrittweise implementiert, bis die Ergebnismenge positiv bewertet wird. In diesem Kapitel werden allerdings nur die letzten *Query* vorgestellt, die auch im Programm verwendet werden. Die restlichen *Query* werden aufgrund ihren Nachteilen nicht eingesetzt aber auch nicht gelöscht, weil sie für die Evaluierungsphase wichtig sind.

Die Controllschicht bietet für die Suche nur eine einzige Methode an und erwartet als Parameter ein von Benutzer ausgewähltes Suchverfahren. Wenn kein ausge-

wählt wird, werden alle Arbeitsangebote einfach nach Datum sortiert ausgegeben.

```
public Page<EsJobDetail> findAndSort(SearchModel searchModel) {
   switch (searchModel.getSearchEngineEnum()) {
      case SIMPLE:
         return findUsingSimpleSearchEngine(searchModel);
      case COMPLEX:
         return findUsingPersonalSearchEngine(searchModel);
      default:
         return findEsPage(searchModel);
   }
}
```

Im Rahmen dieses Kapitel wird auch entsprechende Serviceimplementierung vorgestellt. Da alle Services in ihren Logik sehr ähnlich sind, werden sie nicht extra beschrieben. Die Services werden mit *Spring Rest* realisiert. Dabei muss jede Klasse, die Rest-Services bietet, als *@RestController* annotiert werden.

```
/**
 * JobDetail-Services.
 * <p>
 * Handle all request with endpoints <code>/jobdetails**</code>
 */
@RestController
public class JobDetailServices extends BaseServices {
...
```

Jede Service-Methode muss die Ressource-URL und die HTTP-Methode mit Hilfe von Annotation *@RequestMapping* setzten.

```
/**
* GET-Request: find job details by specified search filter.
*
* @return data transfer object <code>ResponseJobDetailTableData</code>
*/
@RequestMapping(value = "/api/jobdetails/{offset}/{limit}", method = GET)
public @ResponseBody ResponseJobDetailTableData getJobDetails(...){
...
}
```

Die *offset* und *limit* Werte sind *Path*-Variablen und sind ein fester Teil der URL. Diese Werte bestimmen die Anzahl der Elementen in der Ergebnismenge und müssen als Methodenparametern deklariert werden. Die weitere Methodenparameter, *serachFilter* und *searchData*, sind optional und werden deswegen nur gesetzt, wenn sie nicht *null* sind.

```
@PathVariable int offset,
@PathVariable int limit,
```

```
@RequestParam(value = "searchFilter", required = false) String searchFilter,
@RequestParam(value = "searchData", required = false) String searchData
```

Der Rückgabewert ist ein Data Transfer Objekt, das als *JSON* repräsentiert wird. Das Ergebnis wird in Response-Body an Client zurücksendet. Der übergebene Parameter *false* in Konstruktor *ResponseJobDetailTableData* sorgt dafür, dass der Inhalt von Arbeitsangebot nicht gesendet wird. Die komplette Service-Methode befindet sich im *Anhang 4.2: Rest-Service: Suche nach Arbeitsangeboten*.

4.3.1 Volltextsuche

Die Volltextsuche wird mit der booleschen *Query* implementiert. Allerdings sortiert *ElasticSearch* standardmäßig auch die boolesche Ergebnisse nach ihrer Relevanz. Bei der Relevanz-Berechnung werden TF/IDF-Faktoren aus dem Vektorraummodell mit anderen Faktoren wie *coordination* und *field length normalization* kombiniert[Gormley u. a. 2015].

Wenn eine Suchanfrage aus mehreren Wörtern besteht, werden sie mit UND-Operator verknüpft. Das führt dazu, dass ein Dokument erst dann relevant wird, wenn alle gesuchte Worte in dem Dokument vorhanden sind.

```
QueryBuilders.boolQuery().should(QueryBuilders.matchQuery("content",
    searchData.trim()).operator(MatchQueryBuilder.Operator.AND));
```

Wenn eine Suchanfrage aus mehreren durch Komma getrennten Suchbegriffen besteht, wird für jeden Suchbegriff eine boolesche *Query* gebildet. Dadurch entsteht eine *ENTWEDER-ODER* Beziehung und mindestens eine *Query* muss zutreffen, damit das Dokument relevant wird.

```
String[] searchDataArray = searchData.split(",");
BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery();
for (String searchTerm : searchDataArray) {
   QueryBuilder matchQuery = QueryBuilders.matchQuery("content",
      searchTerm.trim()).operator(MatchQueryBuilder.Operator.AND);
   boolQueryBuilder.should(matchQuery);
}
```

Bisher wurde nur das Feld *content* durchgesucht. Laut Anforderungen muss auch das Datum bei der Relevanz-Berechnung berücksichtigt werden. Durch viele Recherche und Ausprobieren wurde die *Decay*-Funktion in ElasticSearch-API entdeckt, die die gestellte Anforderungen erfüllt.

```
FunctionScoreQueryBuilder functionBuilder=QueryBuilders
   .functionScoreQuery(boolQueryBuilder);
functionBuilder.add(ScoreFunctionBuilders
   .exponentialDecayFunction("receivedDate", "14d"));
```

Bei der Implementierung wird eine exponentiale *Decay*-Funktion mit der Distanz von 14 Tagen berechnet. Die Formel zur Berechnung wurde bereits in dem Kapitel 2.5 beschrieben und die einzelne Variablen der Formel können wie folgt ersetzt werden.

- $fieldValue$ - 7.10.2015 - 1444168800000 ms von 1 Januar 1970
- $origin$ - Heute - 28.02.2016 - 1456614000000 ms von 1 Januar 1970
- $scale$ - 14 Tage - 1209600000 ms
- $decay$ - 0.5

Letztendlich wird noch ein Filter eingebaut, das die gelesene Dokumente aus der Suche ausschließt. Die vollständige Implementierung liegt im *Anhang 4.3: Volltext -SearchQuery*.

```
QueryBuilder builder = QueryBuilders.filteredQuery(boolQueryBuilder,
   FilterBuilders.termFilter("read", Boolean.FALSE));
```

4.3.2 Gewichtete Suche

Die gewichtete Suchanfragen werden mit *Boosting* realisiert. Wenn ein Suchbegriff zutrifft, wird die *Query*, die diesen Suchbegriff enthält, mit einem bestimmten Faktor geboosted. Damit ändert sich die Score und auch die Relevanz dieses Dokumentes gegenüber der anderen Dokumenten.

Alle indexierte Benutzerdaten werden geladen und für jedes Index wird eine eigene boolesche *Query* gebildet. Die *Query* wird mit Hilfe von Weight-Score-Funktion geboosted. Im Vergleich zu der einfachen *Boosting*-Verfahren wird die *Query* auch dann geboosted, wenn der Suchterm nicht in exakt gleichen Form in dem Dokument vorkommt.

```
FunctionScoreQueryBuilder weightScoreQuery = QueryBuilders
   .functionScoreQuery(matchQueryBuilder)
   .add(ScoreFunctionBuilders.weightFactorFunction(boostValue));
boolQueryBuilder.should(weightScoreQuery);
```

Der *Boost*-Faktor(*boostValue*) wird auf zwei verschiedene Art und Weise berechnet. Wenn die *Skills* direkt von Benutzer bewertet wurden, werden ihre Bewertungen mit 10 multipliziert und als *Boost*-Faktor verwendet. Das wird gemacht um das Multiplizieren durch 1 zu verhindern, weil ein *Skill* technisch auch mit 1 bewertet werden könnte. Wenn die *Skills*-Rating programmatisch durch die Anzahl von ausgewählten Unterelementen bestimmt wird, wird es nicht mit 10 multipliziert. Damit haben die Elemente, die direkt von Benutzer bewertet wurden, eine größere Einfluss auf die Such-*Query*.

```
float boostValue = esUserSkill.getRating() * 10;
if (esUserSkill.isParent()) {
   // for parent override.
   boostValue = esUserSkill.getRating();
}
```

Für die gewichtete Suche wird auch die *Decay*-Funktion und das *Read*-Filter verwendet. Die Gesamtrelevanz wird durch die Multiplizieren von allen berechneten Score einer *Query* bestimmt. Die vollständige Implementierung liegt im *Anhang 4.4: Boosting-SearchQuery*.

4.3.3 Präfix-Suche

Die Präfix-Suche oder Autovervollständigung wird mit *Completion-Suggest-Query* realisiert. Laut Dokumentation auf der Herstellerseite wird die *Completion-Suggest-Query* genau für diesen Zweck implementiert und bietet im Vergleich zu anderen *Query* eine bessere Performance an. Die *Query* sucht in einem speziell indexierten Feld nur nach den Anfangsbuchstaben und das ist ein wesentlicher Beschleunigungsfaktor.

```
public List<EsAutocompleteLanguage> suggestLanguage(String wordPrefix) {
   CompletionSuggestionFuzzyBuilder completionSuggestionFuzzyBuilder =
      new CompletionSuggestionFuzzyBuilder("suggestLanguage")
         .text(wordPrefix).field("suggestLanguage");
   SuggestResponse suggestResponse = elasticsearchTemplate.suggest(
      completionSuggestionFuzzyBuilder, EsAutocompleteLanguage.class);

   CompletionSuggestion completionSuggestion = suggestResponse.getSuggest()
      .getSuggestion("suggestLanguage");
   List<CompletionSuggestion.Entry.Option> options = completionSuggestion
      .getEntries().get(0).getOptions();

   return options.stream().map(option -> new EsAutocompleteLanguage(
      option.getPayloadAsMap())).collect(Collectors.toList());
}
```

Wie bereits im Kapitel *„Daten Indexierung"* erwähnt wurde, liefert die *Completion-Suggest-Query* nur das vollständige Wort zurück. Deswegen muss noch das mitgespeicherte *Payload* gelesen werden, um das betroffene Objekt eindeutig identifizieren zu können.

4.4 Front-End

Der Front-End-Teil ist eine eigenständige *AngularJS*-Anwendung. In dem Kapitel 3.2 vordefinierten *MockUps* für Benutzerprofile- und Arbeitsangebotsseite werden eins zu eins umgesetzt. Für die Darstellung von allen unterstützten Qualifikationen und Kenntnissen in einem Baumstruktur wird eine *AngularJS*-Direktive[2] entwickelt, die als HTML-Element verwendet werden kann. Um die Benutzerkenntnisse im Baum automatisch selektieren zu können, erwartet das Element zwei List-Attribute mit unterstützen und von Benutzer ausgewählten Qualifikationen.

```
<skill-tree ng-model='supportedPersonalStrengths'
    selected='userPersonalStrengths' expanded='false'></skill-tree>
```

Die Qualifikationsbewertung wird auch als Direktive implementiert. Allerdings kann diese Direktive nur als HTML-Attribut innerhalb eines anderes HTML-Element benutzt werden.

```
<span class='star-rating' star-rating
    rating-value='personalStrength.rating' data-max='5'
   on-rating-selected='rateFunction(rating)'></span>
```

Laut der neuen Anforderungen darf der Benutzer seine Qualifikation erst dann bewerten, wenn die erweiterte Suche als Suchfilter ausgewählt wird. Deswegen wird auch dieses Dialog erst dann angezeigt, wenn der entsprechende Filter ausgewählt wird.

Zur Darstellung von *Tag Clouds* wird die *JQuery-Bibliothek jQCloud*[3] verwendet, die eine Element-Direktive für *AngularJS*-Anwendungen bietet.

```
<jqcloud words="words" height="150" autoResize="true" steps="5"></jqcloud>
```

Der übergebene Parameter *words* ist eine Liste von Objekten, die sowohl ein Text als auch die Gewichtung beinhalten. Das Attribut *steps* repräsentiert die Mapping von Worten abhängig von Ihrer Gewichtung. In der Anwendung ist es der maximale Bewertungswert.

Nachdem der Benutzer sein Profile erfolgreich gespeichert hat, werden einige Benutzerdaten, wie Id, Vorname und Nachname, in Cookies geschrieben. Bei jedem Aufruf der Seite wird es geprüft, ob die Cookies mit Benutzerdaten existiert und erst wenn es nicht der Fall ist, werden die Daten erneut von Server geladen.

[2] **https://docs.angularjs.org/guide/directive**.
[3] **http://mistic100.github.io/jQCloud/index.html**.

5. Evaluierung

Während der Masterarbeit wurden mehrere *Search-Query* implementiert, die aber aufgrund ihren schlechten Effektivität in der Anwendung nicht angewendet werden. Da alle diese *Query* durch schrittweise Optimierung von Nachteilen zu der Endergebnis geführt haben, werden auch diese *Query* in diesem Kapitel evaluiert und detailiert beschrieben.

Um einzelne *Query* mit einander effektiv vergleichen zu können, müssen zwei Voraussetzungen erfüllt werden. Die Evaluierung muss genau auf die *Query* beschränkt werden und die Datenmenge muss unverändert bleiben. Der einzige Grund für Datenänderung kann nur durch inkonsistente Datenmenge entstehen, die zur falschen Evaluierungsergebnissen führt. In der Tabelle unten wird deswegen der Datenausgangszustand kurz zusammengefasst.

Anzahl Dokumenten	Ältestes Datum	Neuestes Datum	Analyzer
3144	14.03.2015	11.10.2015	standard

Tabelle 5.1: Datenausgangszustand für die Evaluierung

Wie in dem vorherigen Kapitel schon erwähnt wurde, wird die Implementierung von allen *SearchQueries* in statische Methode separat ausgelagert und über eine Utility-Klasse(*SearchQueryUtility*) zur Verfügung gestellt. Für die Evaluierung werden diese Methode einzeln ausgeführt und die resultierte *SearchQuery* zusammen mit Pagination-Daten an *ElasticSearch*-Client direkt übergeben. Um die *SearchQuery* besonders die *Score*-Berechnung genau untersuchen zu können, bittet *ElasticSearch* durch *Explain*-Mode eine sehr detailierte Ergebnisausgabe. Mit dem *SpringData-Framework* und *ElasticSearch-API* kann es wie folgt realisiert werden.

```
PageRequest pageRequest = new PageRequest(0, 50);
SearchQuery searchQuery = SearchQueryUtility.buildFullTextSearchBoolQuery_1(
     "Werkstudent Java, Datenbanken", pageRequest);
SearchResponse searchResponse = elasticsearchConfiguration.getNode()
    .client().prepareSearch("intellijob").setTypes("jobDetails")
         .setQuery(searchQuery.getQuery())
         .setExplain(true)
         .setSize(50)
         .get();
```

5.1 Volltext-Suchanfragen

Query Nr. 1

```
{
  "bool": {
    "should": [
      {
        "match": {
          "content": {
            "query": "Werkstudent Java",
            "type": "boolean",
            "operator": "AND"
          }}
      },
      {
        "match": {
          "content": {
            "query": "Datenbanken",
            "type": "boolean"
          }}}
    ],
    "minimum_should_match": "1"
  }
}
```

Beschreibung: Es gibt zwei boolesche Suchanfragen, die zu einer gemeinsamen ebenfalls boolescher Suchanfrage zusammengefasst sind. Die Suchbegriffe sind mit *AND* verknüpft. *„minimum_should_match : 1"* sorgt dafür, dass entweder die erste oder die zweite Suchanfrage treffen muss, damit das Dokument relevant wird.

#	Score	Arbeitsbezeichnung	Datum
1	0.13931298	WERKSTUDENT SOFTWAREENTWICKLUNG IM BEREICH BAHNTECHNIK	21.09.2015 11-24
2	0.13591066	Werkstudent im Bereich Software Entwicklung - Smart HomeBosch Software Innovations GmbH	28.04.2015 02-50
3	0.13591066	Werkstudent im Bereich Software Entwicklung - Smart HomeBosch Software Innovations GmbH	28.04.2015 02-50

Tabelle 5.2: Query Nr. 1 - Ergebnis-Ausschnitt

Ergebnis: Zur Evaluierung vom ersten *Query* werden nur drei erste Ergebnisse angezeigt, die schon deutlich darauf hinweisen, dass die Datenmenge nicht konsis-

tent ist. Es gibt also mehrere gleiche Arbeitsangebote, die auch gleich gewichtet werden.

Query Nr. 1.1

Nachdem die duplizierte Arbeitsangebote gelöscht wurden, sind nur **2292** Arbeitsangebote übrig geblieben.

Ergebnis: Die Tabelle unten zeigt die ersten fünf Arbeitsangebote an.

#	Score	Arbeitsbezeichnung	Datum
1	0.19563562	Werkstudent im Bereich Software Entwicklung - Smart HomeBosch Software Innovations GmbH - Berlin	16.08.2015 02-29
2	0.1758025	Werkstudent im Bereich Softwareentwicklung (Java)	23.08.2015 02-39
3	0.16529423	WERKSTUDENT SOFTWAREENTWICKLUNG IM BEREICH BAHNTECHNIK	21.09.2015 11-24
4	0.15650849	Werkstudent im Bereich Frontend Design	19.09.2015 03-13
5	0.15650849	Werkstudent im Bereich Software Development Toolchain	18.04.2015 02-53

Tabelle 5.3: Query Nr. 1.1 - Ergebnis-Ausschnitt

Um jede Änderung von *Query* effektiv nachvollziehen und bewerten zu können, werden immer die gleiche Suchbegriffe„*Softwareentwickler Java, Testautomatisierung*", für allen Volltextsuchanfragen verwendet. In der Tabelle(5.4) wird die Relevanz von einzelnen Suchbegriffen zusammengefasst.

Suchbegriff	relevante Dokumente
Softwareentwickler	149
Java	305
Softwareentwickler {AND} Java	73
Testautomatisierung	22
Softwareentwickler {AND} Java {AND} Testautomatisierung	6

Tabelle 5.4: Relevanz von Suchbegriffen für die Volltextsuche

Aus der Tabelle(5.4) wird es ersichtlich, dass die exakte Suche nach *„Softwareentwickler Java, Testautomatisierung*" insgesamt 89 (73+22-6) relevante Arbeitsangebote liefern soll.

Query Nr. 1.2

```
{
  "bool": {
    "should": [
      {
        "match": {
          "content": {
            "query": "Softwareentwickler Java",
            "type": "boolean",
            "operator": "AND"
          }}
      },
      {
        "match": {
          "content": {
            "query": "Testautomatisierung",
            "type": "boolean"
          }
        }}
    ],
    "minimum_should_match": "1"
  }
}
```

Ergebnis:

#	Arbeitsbezeichnung	Datum
1	Softwareentwickler RefactoringInterhyp	15.07.2015
2	Softwareentwickler/ -architekt	09.07.2015
3	Senior Java-Entwickler Angebotsimport	01.10.2015
4	Senior Java-Entwickler Full Stack	14.05.2015
5	Senior Software Developer (Java/Backend)	21.09.2015

Tabelle 5.5: Query Nr. 1.2 - Ergebnis- 5 beste Hits

Die Tabelle(5.5) zeigt die ersten fünf Dokumenten der Ergebnisliste. Insgesamt werden genau 89 Arbeitsangebote gefunden. Wenn diese Zahl in die Formel zur Berechnung von *Recall* und *Precision* aus dem Kapitel 2.4.1 eingesetzt wird, kommt als Ergebnis raus:

Ergebnis(D)	Rel. in Ergebnis (a)	Rel. in Dok(S)	Recall	Precision
89	89	89	1 = 100%	1 = 100%

Tabelle 5.6: Query Nr. 1.1 - falsch-positive Recall, Precision

Auf den ersten Blick scheint die *Query* echt positiv zu sein. Denn die Relevanz eines Dokumentes bei der booleschen Suchanfrage wird allein durch das Existenz von gesuchten Termen bestimmt. Wenn man aber die Ergebnismenge aus der Tabelle(5.5) unter Berücksichtigung des Datums von zuletzt gespeicherten Arbeitsangebot am 11.10.2015 genau anschaut, fehlt es auf, dass einige Arbeitsangebote sehr veraltet und aus der Benutzersicht nicht mehr relevant sind.

In Wirklichkeit bei der großen Datenmenge und bei der großen Anzahl von relevanten Dokumenten interessiert es den Benutzer nicht, wieviele Dokumente gefunden werden, sondern viel mehr wie relevant die ersten 10 oder 20 Dokumente sind. Für die Evaluierung von sortierter Ergebnisliste werden *Recal* und *Precision* Werte, wie im Kapitel 2.4.1 beschrieben ist, für jedes Dokument einzeln berechnet. Außerdem muss auch die erwartende Ergebnisliste von 10 relevantesten Dokumenten festgelegt und eingestuft werden.

#	**Arbeitsbezeichnung**	**Datum**
1	Senior Java-Entwickler Angebotsimport	01.10.2015
2	SOFTWAREENTWICKLER	07.10.2015
3	Senior Software Developer (Java/Backend)	21.09.2015
4	Senior Software Developer (Java) – Single Sign-On	21.09.2015
5	IT-Consultant ERP/ Softwareentwickler	07.10.2015
6	Softwareentwickler/in - Berlin	01.10.2015
7	Senior Softwareentwickler/-architekt Java EE	01.10.2015
8	Web-Entwickler	01.10.2015
9	Softwareentwickler Java für Versicherungen	30.09.2015
10	Webentwickler Frontend	27.09.2015

Tabelle 5.7: Die erwartende Ergebnismenge für Volltextsuche

Um die Testmenge zu bestimmen, werden 3 booleschen Suchanfragen ausgeführt und nach Datum sortiert ausgegeben. Die Arbeitsangebote *„Senior Java-Entwickler Angebotsimport*", *„Senior Software Developer (Java/Backend)*" und *„Senior Software Developer (Java) – Single Sign-On*" beinhalten sowohl *„Softwareentwickler Java*" als auch *„Testautomatisierung*" Terme. Das Arbeitsangebot *„Webentwickler Frontend*" ist das einzige Dokument, das nur den Term *„Testautomatisierung*" enthält. Die restliche Arbeitsangebote sind die Ergebnisse für die Suchanfrage *„Softwareentwickler Java*". Bei der Festlegung von Position innerhalb der erwartenden Ergebnisliste wurde auf die Anzahl von gesuchten Termen und das Datum geachtet. Unter Berücksichtigung der neuen Testmenge können *Recall* und *Precision* für 10 ersten Hits der ersten *Query* berechnet werden.

#	Arbeitsbezeichnung	Datum	Rel. Dok. bisher	Rec.	Prec.
1	Softwareentwickler RefactoringInterhyp AG	15.07.2015	0	0	0
2	Softwareentwickler/ -architekt	09.07.2015	0	0	0
3	Senior Java-Entwickler Angebotsimport	01.10.2015	1	0.1	0.33
4	Senior Java-Entwickler Full Stack	14.05.2015	1	0.1	0.25
5	Senior Software Developer (Java/-Backend)	21.09.2015	2	0.2	0.4
6	Senior Software Developer (Java) – Single Sign-On	21.09.2015	3	0.3	0.5
7	Spezialist Testautomatisierung	10.07.2015	3	0.3	0.43
8	Softwareentwickler	18.05.2015	3	0.3	0.38
9	Java-Softwareentwickler	08.05.2015	3	0.3	0.33
10	Java Softwareentwickler	21.05.2015	3	0.3	0.3

Tabelle 5.8: Query Nr. 1.2 - 10 beste Hits

Die zwei erste Hits in der Ergebnisliste gehören nicht zu den erwartenden Arbeitsangeboten und ihre *Recall* und *Precision* sind deswegen 0. Das Arbeitsangebot *„Senior Java-Entwickler Angebotsimport*" wird erwartet und das ist auch das erste gefundene Dokument.

$$Recall = \frac{relevanteDokumenteBisher}{AnzahlErwartendenDokumenten} \Longrightarrow \frac{1}{10} = 0.1$$

$$Precision = \frac{relevanteDokumenteBisher}{AnzahlDokumentenBisher} \Longrightarrow \frac{1}{3} = 0.33$$

Das nächste Arbeitsangebot wird wieder nicht erwartet. Deswegen bleibt auch seine *Recall* unverändert. Der *Precision* Wert wird kleiner $1/4 = 0.25$. Die restliche *Recall* und *Precision* Werte werden analog zu dem Beispiel berechnet. Nachdem alle *Recall* und *Precision* Werte bekannt wurden, kann auch *interpolated precision* für jeden *Recall*-Punkt berechnet wurden.

Recall	0.1	0.2	0.3
interpolated precision	0.33	0.4	0.5

Tabelle 5.9: Query Nr. 1.2 - Interpolated precision

Query Nr. 2

```
{
  "bool": {
    "should": [
      {
        "match": {
          "content": {
            "query": "Softwareentwickler Java",
            "type": "boolean",
            "operator": "AND"
          }}
      },
      {
        "match": {
          "content": {
            "query": "Testautomatisierung",
            "type": "boolean"
          }
        }}
    ],
    "minimum_should_match": "1"
  },
  "sort": [ { "receivedDate": { "order": "desc"}}]
}
```

Beschreibung: Die zweite *Query* sortiert einfach die Ergebnisliste absteigend nach dem Feld *receivedDate*, das als Datum repräsentiert wird.

Ergebnis: Die Tabelle(5.10) zeigt, dass die Ergebnismenge jetzt absteigend nach Datum sortiert ist. Die folge von *Recall* und *Interpolated precision* sind jetzt im Vergleich zu der alten Suchanfrage viel besser geworden.

Interessant ist auch der *Explain*-Plan von dieser *Query*, dessen Ausschnitt im Anhang (siehe Anhang 5.1: Query Nr. 2 - Explain-Plan des ersten Hits) zu finden ist. Die eigentliche *Score*-Berechnung für den ersten Hit ist **0.12759583** wird aber durch **NaN** ersetzt. **NaN** steht in *ElasticSearch* für maximale *Score*. Das heißt, dass die Dokumente, die eigentlich für die Suchanfrage nicht wirklich relevant sein könnten, durch Sortierverfahren und das Maximale *Score* jedoch relevant werden. Daraus folgt, dass die *Score* bei der Sortierung nicht mehr berücksichtigt wird.

#	Arbeitsbezeichnung	Datum	Rel. Dok. bisher	Rec.	Prec.
1	SOFTWAREENTWICKLER	07.10.2015	1	0.1	1
2	IT-Consultant ERP/ Softwareentwickler	07.10.2015	2	0.2	1
3	Softwareentwickler/in - Berlin	01.10.2015	3	0.3	1
4	Senior Softwareentwickler/-architekt Java EE	01.10.2015	4	0.4	1
5	Web-Entwickler	01.10.2015	5	0.5	1
6	Senior Java-Entwickler Angebotsimport	01.10.2015	6	0.6	1
7	Softwareentwickler Java für Versicherungen	30.09.2015	7	0.7	1
8	Softwareentwickler (Java) für Banken	30.09.2015	7	0.7	0.88
9	Hochschulabsolvent - Softwareentwickler Java	30.09.2015	7	0.7	0.78
10	Softwareentwickler Lotus Notes	29.09.2015	7	0.7	0.7

Tabelle 5.10: Query Nr. 2 - 10 beste Hits

Recall	0.1	0.2	0.3	0.4	0.5	0.6	0.7
interpolated precision	1	1	1	1	1	1	1

Tabelle 5.11: Query Nr. 2 - Interpolated precision

Query Nr. 3

```
{
  "function_score": {
    "query": {
      "bool": {
        "should": [
          {
            "match": {
              "content": {
                "query": "Softwareentwickler Java",
                "type": "boolean",
                "operator": "AND"
              }}
          },
          {
            "match": {
              "content": {
                "query": "Testautomatisierung",
```

```
            "type": "boolean",
            "operator": "AND"
          }
        }}
      ]}
    },
    "functions": [
      {
        "exp": {
          "receivedDate": {
            "scale": "14d",
            "decay": 0.5
          }
        }}
    ]}
}
```

Beschreibung: Die dritte *Query* ersetzt die Sortierung durch die *Decay*-Funktion (siehe Kapitel 2.5), um das Datum bei der *Score*-Berechnung zu berücksichtigen. Das Datum-Score wird jetzt *exponential* mit dem festgelegten *Scale*-Wert von 14 Tagen berechnet.

Ergebnis:

#	Arbeitsbezeichnung	Datum	Rel. Dok. bisher	Rec.	Prec.
1	Senior Java-Entwickler Angebotsimport	01.10.2015	1	0.1	1
2	Senior Software Developer (Java/-Backend)	21.09.2015	2	0.2	1
3	Senior Software Developer (Java) – Single Sign-On	21.09.2015	3	0.3	1
4	Softwareentwickler/in - Berlin	01.10.2015	4	0.4	1
5	Senior Softwareentwickler/-architekt Java EE	01.10.2015	5	0.5	1
6	Softwareentwickler Java für Versicherungen	30.09.2015	6	0.6	1
7	SOFTWAREENTWICKLER	07.10.2015	7	0.7	1
8	Hochschulabsolvent - Softwareentwickler Java	30.09.2015	7	0.7	0.88
9	Softwareentwickler (Java) für Banken	30.09.2015	7	0.7	0.78
10	Java Experte/Softwareentwickler	24.09.2015	7	0.7	0.7

Tabelle 5.12: Query Nr. 3 - 10 beste Hits

Der *Explain*-Plan im Anhang(Anhang 5.2: Query Nr. 3 - Explain-Plan des ersten 3 Hits) zeigt schon deutlich, dass das Datum bei der *Score* mitgerechnet wird. Mit der aktuellen Testmenge sind *Recall* und *Interpolated precision* unverändert geblieben. Die Reihenfolge von Dokumenten entspricht jetzt aber mehr den Erwartungen.

Recall	0.1	0.2	0.3	0.4	0.5	0.6	0.7
interpolated precision	1	1	1	1	1	1	1

Tabelle 5.13: Query Nr. 3 - Interpolated precision

Query Nr. 4

```
{
  "function_score": {
    "query": {
      "bool": {
        "must": {
          "term": {
            "read": false
          }
        },
        "should": [
          {
            "match": {
              "content": {
                "query": "Softwareentwickler Java",
                "type": "boolean",
                "operator": "AND"
              }}
          },
          {
            "match": {
              "content": {
                "query": "Testautomatisierung",
                "type": "boolean",
                "operator": "AND"
              }}}
        ]}
    },
    "functions": [
      {
        "exp": {
          "receivedDate": {
            "scale": "14d",
            "decay": 0.5
          }}}
    ]}
}
```

Beschreibung: Die *Query* Nr. 4 erweitert die letzte *Query* mit einer neuen booleschen *Term-Query*, die dafür sorgt, dass nur ungelesene Arbeitsangebote durchgesucht werden.

Ergebnis: Interessanteweise hat diese winzige Änderung eine neue Ergebnisliste verursacht, was nicht erwartet und auch nie gewünscht wurde.

#	Arbeitsbezeichnung	Datum	Rel. Dok. bisher	Rec.	Prec.
1	Senior Java-Entwickler Angebotsimport	01.10.2015	1	0.1	1
2	SOFTWAREENTWICKLER	07.10.2015	2	0.2	1
3	Softwareentwickler/in - Berlin	01.10.2015	3	0.3	1
4	Senior Softwareentwickler/-architekt Java EE	01.10.2015	4	0.4	1
5	Senior Software Developer (Java/-Backend)	21.09.2015	5	0.5	1
6	Senior Software Developer (Java) – Single Sign-On	21.09.2015	6	0.6	1
7	Softwareentwickler Java für Versicherungen	30.09.2015	7	0.7	1
8	Hochschulabsolvent - Softwareentwickler Java	30.09.2015	7	0.7	0.88
9	IT-Consultant ERP/ Softwareentwickler	07.10.2015	8	0.8	0.89
10	Softwareentwickler (Java) für Banken	30.09.2015	8	0.8	0.8

Tabelle 5.14: Query Nr. 4 - 10 beste Hits

Recall	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
interpolated precision	1	1	1	1	1	1	1	0.89

Tabelle 5.15: Query Nr. 4 - Interpolated precision

Aus der Tabelle oben folgt, dass die neue Ergebnismenge im Vergleich zu dem alten Ergebnis besser geworden ist. Die Untersuchung von *Explain*-Plan(Anhang 5.3: Query Nr. 4 - Explain-Plan des ersten Hit) hat gezeigt, dass die neue boolesche Term-*Query* bei der *Score*-Berechnung mitgezählt wird. Das darf aber nicht passieren. Das Feld *read* ist ein Flag, der ein Dokument aus dem Suchvorgang komplett ausschliessen soll und keinen Einfluss auf seine Relevanz haben darf.

Query Nr. 5

```
{
  "function_score" : {
    "query" : {
      "filtered" : {
        "query" : {
          "bool" : {
            "should" : [ {
              "match" : {
                "content" : {
                  "query" : "Softwareentwickler Java",
                  "type" : "boolean",
                  "operator" : "AND"
                }}
            }, {
              "match" : {
                "content" : {
                  "query" : "Testautomatisierung",
                  "type" : "boolean",
                  "operator" : "AND"
                }}
            } ]
          }
        },
        "filter" : {
          "term" : {
            "read" : false
          }}}
    },
    "functions" : [ {
      "exp" : {
        "receivedDate" : {
          "scale" : "14d"
        }}
    } ]
  }
}
```

Beschreibung: Die fünfte und auch die letze *Query* im Zusammenhang mit der Volltextsuche verwendet einen Filter statt eine boolesche Term-Anfrage, um das Statusfeld *read* anzufragen. Der Filter kann jetzt die Dokumente nach dem *Read*-Status ohne Einfluss auf ihre Relevanz filtern.

Ergebnis: Die Ergebnismenge wird hier nicht extra aufgelistet, weil sie äquivalent zu dem Ergebnis der dritten *Query* ist. Der *Explain*-Plan des ersten Hits befindet sich im Anhang(Anhang 5.4: Query Nr. 5 - Explain-Plan des ersten Hits).

5.2 Gewichtete Suchanfragen

Die gewichtete Suchanfrage verwendet als Suchterme die persönlichen Fähigkeiten und Qualifikationen, die der Benutzer selbst eingeschätzt hat. Diese Einschätzung muss jedenfalls bei der Suche berücksichtigt werden. Damit es sinnvoll getestet werden könnte, muss die Suchanfrage aus mehreren Suchbegriffen mit unterschiedlicher Einschätzung bestehen. Die *Query*-Struktur wird dadurch größer und wird deswegen in den Anhang ausgelagert.

Die Testszenarien beinhaltet vier unterschiedlich geschätzten Suchbegriffe.

Qualifikation	**Einschätzung**
Spring Framework	5
Java	3
PostgreSQL	2
Datenbanken	2

Tabelle 5.16: Einschätzung von Qualifikationen für die Evaluierung

Die Tabelle(5.17) zeigt auch die Anzahl von relevanten Dokumenten für die oben aufgelistete Suchbegriffe.

Suchbegriff	**relevante Dokumente**
Spring	151
Framework	136
Spring {AND} Framework	43
Java	305
PostgreSQL	17
Datenbanken	171
Spring {AND} Framework {AND} Java {AND} PostgreSQL {AND} Datenbanken	2

Tabelle 5.17: Relevanz von Suchbegriffen für die gewichtete Suche

Die erwartende Ergebnismenge wird in der Tabelle(5.18) zusammengefasst. Bei der Relevanzbestimmung wird auf das Vorkommen von Suchbegriffen in Abhängigkeit von ihrer Schätzung geachtet.

#	Arbeitsbezeichnung	Datum
1	Java Software Developer Connected Mobility	01.10.2015
2	Senior Java-Entwickler Angebotsimport	01.10.2015
3	Softwareentwickler (Java) für Banken	30.09.2015
4	Softwareentwickler Java für Versicherungen	30.09.2015
5	Java Developer! Inhouse Java Software Entwickler	28.09.2015
6	Software Developer (Java/Backend)	21.09.2015
7	Software Developer DWH	27.09.2015
8	Java Developer JEE / J2EE	03.10.2015
9	Junior Datenbank Analyst-Bereich Reise	02.10.2015
10	Softwareentwickler (Java) für Banken	30.09.2015

Tabelle 5.18: Die erwartende Ergebnismenge für die gewichtete Suche

Query Nr. 6

Beschreibung: Die 6. *Query*(Anhang 5.5: Query Nr. 6) benutzt die *Boost*-Funktion, um die Score-Wert für einen bestimmten Suchterm mit dem *Boost factor* zu multiplizieren. Die *Boost factor* wird nach der Formel $Factor = Einschaetzung * 10$ berechnet.

Ergebnis: Die *Query* findet nur fünf relevante Dokumente. Da die relevanteste Dokumente jedoch gefunden werden, wird die Gesamtrelevanz der Ergebnismenge trotzdem positiv bewertet. Aus dem *Explain*-Plan(Anhang 5.6: Query Nr. 6 - Explain-Plan des ersten Hit) wird es ersichtlich, dass nur ein exakter Suchtreffer geboosted wird. So werden nur die Begriffe, *Java* und *Datenbank*, aber nicht *Spring Framework* geboosted.

#	Arbeitsbezeichnung	Datum	Rel. Dok. bisher	Rec.	Prec.
1	Senior Java-Entwickler Angebotsimport	01.10.2015	1	0.1	1
2	Softwareentwickler Java für Versicherungen	30.09.2015	2	0.2	1
3	Software Developer DWH	27.09.2015	3	0.3	1
4	Java Software Developer Connected Mobility	01.10.2015	4	0.4	1
5	Java Developer! Inhouse Java Software Entwickler	28.09.2015	5	0.5	1
6	Software Developer (PHP / Full Stack)	01.10.2015	5	0.5	0.83
7	Java Backend Entwickler	28.09.2015	5	0.5	0.71
8	Junior Softwareentwickler/in Java	07.09.2015	5	0.5	0.63
9	Senior Software Developer (Java)	21.09.2015	5	0.5	0.56
10	Java Frontend Developer	19.09.2015	5	0.5	0.5

Tabelle 5.19: Query Nr. 6 - 10 beste Hits

Recall	0.1	0.2	0.3	0.4	0.5
interpolated precision	1	1	1	1	1

Tabelle 5.20: Query Nr. 6 - Interpolated precision

Query Nr. 7

Beschreibung: Die letzte *Query*(Anhang 5.7: Query Nr. 7) verknüpft alle Suchterme mit *OR* und verwendet eine zusätzliche *Weight*-Funktion, um die Score noch einmal zu boosten. Im Zusammenhang mit *OR*-Verknüpfung werden jetzt alle Dokumente mit dem *Spring*- oder *Framework*-Term höher gewichtet.

Ergebnis: Obwohl die Ergebnisliste sich für 10 ersten Hits kaum geändert hat, ist die *Query* insgesamt besser geworden. So wird beispielsweise die Position des Arbeitsangebotes *„Softwareentwickler (Java) für Banken*" um 10 Stellen(von 19 zu 9) geändert. Die *Query* hat allerdings auch einen Nachteil und zwar die Arbeitsangebote, wo das Wort *Framework* nicht im Zusammenhang mit *Spring* sondern mit *Play* oder *AngularJS* vorkommt, ebenfalls geboosted werden.

#	Arbeitsbezeichnung	Datum	Rel. Dok. bisher	Rec.	Prec.
1	Senior Java-Entwickler Angebotsimport	01.10.2015	1	0.1	1
2	Softwareentwickler Java für Versicherungen	30.09.2015	2	0.2	1
3	Software Developer DWH	27.09.2015	3	0.3	1
4	Java Software Developer Connected Mobility	01.10.2015	4	0.4	1
5	Java Developer! Inhouse Java Software Entwickler	28.09.2015	5	0.5	1
6	Software Developer (PHP / Full Stack)	01.10.2015	5	0.5	0.83
7	Java Backend Entwickler	28.09.2015	5	0.5	0.71
8	Senior Software Developer (Java)	21.09.2015	5	0.5	0.63
9	Softwareentwickler (Java) für Banken	30.09.2015	6	0.6	0.68
10	Junior Softwareentwickler/in Java	07.09.2015	6	0.6	0.6

Tabelle 5.21: Query Nr. 7 - 10 beste Hits mit der Relevanzeinstufung

Recall	0.1	0.2	0.3	0.4	0.5	0.6
interpolated precision	1	1	1	1	1	0.68

Tabelle 5.22: Query Nr. 7 - Interpolated precision

5.3 Vergleich und Fazit

Um Volltextsuche mit der gewichteten Suche sinnvoll vergleichen zu können, werden die *Query* Nr.5 und die *Query* Nr. 7 mit den gleichen Suchtermen(siehe Tabelle 5.16) ausgeführt. Aus der Tabelle(5.23) kann man erschließen, dass die Werte für *Recall* und *Interpolated precision* der Volltext-Suchanfrage äquivalent zu den Werten der 6. Query sind. Die Untersuchung von Arbeitsangeboten *„Senior Software Developer (Java)*" und *„Softwareentwickler (Java) für Banken*" aus der gewichteten Suchanfrage hat ergeben, dass diese Arbeitsangebote aus der Benutzersicht viel relevanter als die Arbeitsangebote aus der Volltextsuche (*„(Senior) PHP-Entwickler*", *„Software Engineer Backend Zalando*") sind. Wenn man die Recall, *Interpolated precision* und Benutzerbewertung zusammenfasst, kommt man zu dem Ergebnis, dass die gewichte Suche im Vergleich zur Volltextsuche insgesamt mehr relevante Dokumente liefert.

#	Query	Arbeitsbezeichnung	Datum	Rel. Dok. bisher	Rec.	Prec.
1	5	Senior Java-Entwickler Angebotsimport	01.10.2015	1	0.1	1
1	7	Senior Java-Entwickler Angebotsimport	01.10.2015	1	0.1	1
2	5	Softwareentwickler Java für Versicherungen	30.09.2015	2	0.2	1
2	7	Softwareentwickler Java für Versicherungen	30.09.2015	2	0.2	1
3	5	Software Developer DWH	27.09.2015	3	0.3	1
3	7	Software Developer DWH	27.09.2015	3	0.3	1
4	5	Java Software Developer Connected Mobility	01.10.2015	4	0.4	1
4	7	Java Software Developer Connected Mobility	01.10.2015	4	0.4	1
5	5	Java Developer! Inhouse Java Software Entwickler	28.09.2015	5	0.5	1
5	7	Java Developer! Inhouse Java Software Entwickler	28.09.2015	5	0.5	1
6	5	Software Developer (PHP / Full Stack)	01.10.2015	5	0.5	0.83
6	7	Software Developer (PHP / Full Stack)	01.10.2015	5	0.5	0.83
7	5	Java Backend Entwickler	28.09.2015	5	0.5	0.71
7	7	Java Backend Entwickler	28.09.2015	5	0.5	0.71
8	5	(Senior) PHP-Entwickler	21.09.2015	5	0.5	0.63
8	7	Senior Software Developer (Java)	21.09.2015	5	0.5	0.63
9	5	Software Engineer Backend Zalando	30.09.2015	5	0.5	0.56
9	7	Softwareentwickler (Java) für Banken	30.09.2015	6	0.6	0.68
10	5	Junior Softwareentwickler/in Java	07.09.2015	5	0.5	0.5
10	7	Junior Softwareentwickler/in Java	07.09.2015	6	0.6	0.6

Tabelle 5.23: Vergleich Query Nr. 5 und Query Nr. 7

6. Problemen und Schwierigkeiten

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

7. Zusammenfassung und Ausblick

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Literaturverzeichnis

Dorschel, Joachim (2015). „Praxishandbuch Big Data - Wirtschaft – Recht – Technik“. 1. Aufl. Berlin Heidelberg New York: Springer-Verlag (siehe S. 4, 6).

Jan, Steemann (2013). *Datenmodellierung in nicht relationalen Datenbanken*. url: https://entwickler.de/online/datenbanken/datenmodellierung-in-nicht-relationalen-datenbanken-137872.html (siehe S. 6).

OnPage (2016). *Tag Cloud*. url: https://de.onpage.org/wiki/Tag_Cloud (siehe S. 12).

Henrich, Andreas (2008). „Information Retrieval Grundlagen, Modelle und Anwendungen“. 1. Aufl. Otto-Friedrich-Universität Bamberg (siehe S. 12 f.).

Manning, Christopher D.; Raghavan, Prabhakar; Schütze, Hinrich (2008). „Introduction to Information Retrieval“. Anniversary. Cambridge: Cambridge University Press (siehe S. 14).

Mandl, Thomas (2001). „Tolerantes Information Retrieval“. Hochschulverband für Informatinswissenschaft(HI)e.V (siehe S. 15).

Salton, G.; Wong, A.; Yang, C. S. (1975). „A Vector Space Model for Automatic Indexing“. *Commun. ACM* 18.11, S. 613–620. doi: 10.1145/361219.361220. url: http://doi.acm.org/10.1145/361219.361220 (siehe S. 16).

Rafal Kuc, Marek Rogozinski (2013). „Mastering ElasticSearch“. 1. Aufl. Packt Publishing Ltd. (siehe S. 20).

Terrier, Francois (2013). *On ElasticSearch performance*. url: https://blog.liip.ch/archive/2013/07/19/on-elasticsearch-performance.html (siehe S. 21).

Brad, Severtson (2015). *Entwicklung und Auswahl von Features in Azure Machine Learning*. url: https://azure.microsoft.com/de-de/documentation/articles/machine-learning-feature-selection-and-engineering (siehe S. 22).

Vladimir Gitlevich, Dan Bergh Johnsson (2006). „Domain-Driven Design Quickly“ (siehe S. 28).

Richardson, Leonard; Ruby, Sam (2008). „RESTful Web Services -". Sebastopol: Ö'Reilly Media, Inc." (siehe S. 31).

Gormley, Clinton; Tong, Zachary (2015). „Elasticsearch: The Definitive Guide -". 1. Aufl. Sebastopol: O'Reilly (siehe S. 42).

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

Chunker - Machine Learning Technologie - Analyse und Zerlegung von Sätzen in Blöcke wie Nomen, Gruppen etc...

Coreference Resolution - Machine Learning Technologie - Bezugnahme auf dieselbe Entität.

DAO - Data Access Object - Adapter zur Abstrahierung und Entkopplung von Datenzugriffe.

Invertierter Index - wird in Suchserver eingesetzt und bezieht sich auf die Tatsache, dass nicht die Dokumente auf Worte zeigen, sondern eine Liste von Worten auf Dokumente zeigt.

IR - Information Retrieval

Named Entity Recognition - Machine Learning Technologie - erkennt und klassifiziert Bestandteile im Text.

NoSQL - not only SQL - bezeichnet Datenbanken, die einen nicht-relationalen Ansatz verfolgen.

Part-Of-Speech tagging - Machine Learning Technologie - Das Zuweisen von Markierungen zu einzelnen Einheiten (Wortart-Annotierung).

SentenceDetector - Machine Learning Technologie - teilt Text in einzelne Sätze

Tokenizer - Machine Learning Technologie - teilt Sätze in einzelne Worte

Anhang

Anhang 4.1: ElasticsearchConfiguration.class

```
/**
 * ElasticSearch Configuration class.
 */
@Configuration
@EnableElasticsearchRepositories(basePackages =
    "com.intellijob.elasticsearch.repository")
public class ElasticsearchConfiguration {

   private static final Logger LOG =
       Logger.getLogger(ElasticsearchConfiguration.class);

   @Value("${elasticsearch.host}")
   private String host;

   @Value("${elasticsearch.clusterName}")
   private String clusterName;

   @Value("${elasticsearch.http.enabled}")
   private boolean httpEnabled;

   @Value("${elasticsearch.http.cors.enabled}")
   private boolean httpCorsEnabled;

   @Value("${elasticsearch.path.data}")
   private String pathData;

   private Node node;

   /**
    * Init elasticsearch client and open connection to cluster.
    *
    * @return default ElasticsearchTemplate connected to cluster.
    */
   @Bean
   public ElasticsearchTemplate elasticsearchTemplate() {
      LOG.info("Start elasticsearch server");
      NodeBuilder nodeBuilder = new NodeBuilder();
```

```
        nodeBuilder.clusterName(clusterName).local(false);
        if (StringUtils.hasLength(pathData)) {
            nodeBuilder.settings().put("path.data", pathData);
        }
        nodeBuilder.settings().put("http.enabled", httpEnabled);
        nodeBuilder.settings().put("http.cors.enabled", httpCorsEnabled);
        nodeBuilder.settings().put("network.host", host);

        node = nodeBuilder.node();
        return new ElasticsearchTemplate(node.client());
    }

    /**
     * Close connection to cluster!
     */
    @PreDestroy
    void destroy() {
        if (node != null) {
            LOG.info("Destroy elasticsearch server.");
            try {
                node.close();
            } catch (Exception e) {
                LOG.error("Elasticsearch server can not be destroyed!", e);
            }
        }
    }

    public Node getNode() {
        return node;
    }
}
```

Anhang 4.2: Rest-Service: Suche nach Arbeitsangeboten

```
/**
* GET-Request: find job details by specified search filter.
*
* @return data transfer object <code>ResponseJobDetailTableData</code>
*/
@RequestMapping(value = "/api/jobdetails/{offset}/{limit}",
                method = RequestMethod.GET)
public @ResponseBody ResponseJobDetailTableData getJobDetails(
@PathVariable int offset, @PathVariable int limit,
@RequestParam(value = "searchFilter", required = false) String searchFilter,
@RequestParam(value = "searchData", required = false) String searchData)
         throws UserNotFoundException
{
   User user = userController.getUniqueUser();

   SearchModelBuilder searchModelBuilder = new SearchModelBuilder(user)
      .setOffset(offset).setLimit(limit);
   if (searchFilter != null) {
      searchModelBuilder.setSearchEngine(searchFilter);
   }
   if (searchData != null) {
     searchModelBuilder.setSearchData(searchData);
   }

   SearchModel searchModel = searchModelBuilder.build();
   Page<EsJobDetail> jobDetailsPage =
       jobDetailController.findAndSort(searchModel);
   return new ResponseJobDetailTableData(jobDetailsPage, Boolean.FALSE);
}
```

Anhang 4.3: Volltext-SearchQuery

```
/**
* Build simple SearchQuery for full text searching.
* <p>
* The searchOriginData will be split by comma and link with OR.
* The search data separated with whitespace will be linked with AND.
* This query use
* <code>ScoreFunctionBuilders.exponentialDecayFunction</code> to calculate
* receivedDate of last 14 days.
*
* @param searchData search data.
* @param pageRequest paging data (offset,limit)
*
* @return build search query
*/
public static SearchQuery buildFullTextQuery(String searchData,
    PageRequest pageRequest)
{
    String[] searchDataArray = searchData.split(",");
    BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery();
    if (searchDataArray.length == 1) {
       boolQueryBuilder.should(QueryBuilders.matchQuery("content",
           searchData.trim()).operator(MatchQueryBuilder.Operator.AND));
    } else {
       for (String searchTerm : searchDataArray) {
          QueryBuilder matchQuery = QueryBuilders.matchQuery("content",
              searchTerm.trim()).operator(MatchQueryBuilder.Operator.AND);
             boolQueryBuilder.should(matchQuery);
       }
    }

    QueryBuilder builder = QueryBuilders.filteredQuery(boolQueryBuilder,
             FilterBuilders.termFilter("read", Boolean.FALSE));
    FunctionScoreQueryBuilder functionBuilder =
        QueryBuilders.functionScoreQuery(builder);
    functionBuilder.add(ScoreFunctionBuilders
             .exponentialDecayFunction("receivedDate", "14d"));

    return new NativeSearchQueryBuilder().withQuery(functionBuilder)
       .withPageable(pageRequest).build();
}
```

Anhang 4.4: Boosting -SearchQuery.

```
/**
* Build a rating search query.
*
* @param skills list of user skills with rating.
* @param offset list offset.
* @param limit list limit.
*
* @return searchQuery
*/
public static SearchQuery buildRatingQuery(Collection<EsUserSkills> skills,
    int offset, int limit) {
   BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery();
   for (EsUserSkills esUserSkill : skills) {
      float boostValue = esUserSkill.getRating() * 10;
      if (esUserSkill.isParent()) {
         // for parent override.
         boostValue = esUserSkill.getRating();
      }
      QueryBuilder matchQueryBuilder = QueryBuilders.matchQuery("content",
         esUserSkill.getName()).operator(MatchQueryBuilder.Operator.OR).boost(boostValue);

      FunctionScoreQueryBuilder weightScoreQuery = QueryBuilders
         .functionScoreQuery(matchQueryBuilder)
         .add(ScoreFunctionBuilders.weightFactorFunction(boostValue));
      boolQueryBuilder.should(weightScoreQuery);
   }

   FilteredQueryBuilder filteredQuery = QueryBuilders
      .filteredQuery(boolQueryBuilder, FilterBuilders
         .termFilter("read", Boolean.FALSE));
   FunctionScoreQueryBuilder functionBuilder =
       QueryBuilders.functionScoreQuery(filteredQuery);
   functionBuilder.add(ScoreFunctionBuilders
            .exponentialDecayFunction("receivedDate", "14d"));

   return buildNativeQuery(functionBuilder, offset, limit);
}
```

Anhang 5.1: Query Nr. 2 - Explain-Plan des ersten Hits

```
1 Hit: score(NaN) | receivedDate(07.10.2015 11-24) |
    name(SOFTWAREENTWICKLER (M/W)) | id(5672646344aeeb036a01be0a)
1 Explain(: 0.12759583 = (MATCH) product of:
  0.25519165 = (MATCH) sum of:
    0.25519165 = (MATCH) sum of:
      0.19744588 = (MATCH) weight(content:softwareentwickl in 364)
          [PerFieldSimilarity], result of:
        0.19744588 = score(doc=364,freq=5.0), product of:
          0.50549334 = queryWeight, product of:
            3.7265449 = idf(docFreq=149, maxDocs=2292)
            0.13564666 = queryNorm
          0.39060035 = fieldWeight in 364, product of:
            2.236068 = tf(freq=5.0), with freq of:
              5.0 = termFreq=5.0
            3.7265449 = idf(docFreq=149, maxDocs=2292)
            0.046875 = fieldNorm(doc=364)
      0.057745762 = (MATCH) weight(content:java in 364)
          [PerFieldSimilarity], result of:
        0.057745762 = score(doc=364,freq=1.0), product of:
          0.40878406 = queryWeight, product of:
            3.0135949 = idf(docFreq=305, maxDocs=2292)
            0.13564666 = queryNorm
          0.14126226 = fieldWeight in 364, product of:
            1.0 = tf(freq=1.0), with freq of:
              1.0 = termFreq=1.0
            3.0135949 = idf(docFreq=305, maxDocs=2292)
            0.046875 = fieldNorm(doc=364)
  0.5 = coord(1/2)
 )
```

Anhang 5.2: Query Nr. 3 - Explain-Plan des ersten 3 Hits

```
1 Hit: score(4.9161434E-4) | receivedDate(01.10.2015 02-55) | name(Senior
    Java-Entwickler Angebotsimport (m/w)Idealo Internet GmbH - Berlin) |
    id(567264c744aeeb036a01be31)
1 Explain(: 4.916144E-4 = (MATCH) function score, product of:
 0.431067 = (MATCH) sum of:
  0.19829285 = (MATCH) sum of:
   0.103017226 = (MATCH) weight(content:softwareentwickl in 403)
       [PerFieldSimilarity], result of:
     0.103017226 = score(doc=403,freq=1.0), product of:
      0.50549334 = queryWeight, product of:
       3.7265449 = idf(docFreq=149, maxDocs=2292)
       0.13564666 = queryNorm
      0.20379542 = fieldWeight in 403, product of:
       1.0 = tf(freq=1.0), with freq of:
        1.0 = termFreq=1.0
       3.7265449 = idf(docFreq=149, maxDocs=2292)
       0.0546875 = fieldNorm(doc=403)
   0.09527563 = (MATCH) weight(content:java in 403) [PerFieldSimilarity],
       result of:
     0.09527563 = score(doc=403,freq=2.0), product of:
      0.40878406 = queryWeight, product of:
       3.0135949 = idf(docFreq=305, maxDocs=2292)
       0.13564666 = queryNorm
      0.23307082 = fieldWeight in 403, product of:
       1.4142135 = tf(freq=2.0), with freq of:
        2.0 = termFreq=2.0
       3.0135949 = idf(docFreq=305, maxDocs=2292)
       0.0546875 = fieldNorm(doc=403)
  0.23277412 = (MATCH) weight(content:testautomatisierung in 403)
      [PerFieldSimilarity], result of:
    0.23277412 = score(doc=403,freq=1.0), product of:
     0.75984997 = queryWeight, product of:
      5.601686 = idf(docFreq=22, maxDocs=2292)
      0.13564666 = queryNorm
     0.3063422 = fieldWeight in 403, product of:
      1.0 = tf(freq=1.0), with freq of:
       1.0 = termFreq=1.0
      5.601686 = idf(docFreq=22, maxDocs=2292)
      0.0546875 = fieldNorm(doc=403)
 0.0011404593 = (MATCH) Math.min of
  0.0011404593 = (MATCH) Function for field receivedDate:
   0.0011404593 = exp(- MIN[Math.max(Math.abs(1.443660906E12(=doc value)
       - 1.455486160446E12(=origin))) - 0.0(=offset), 0)] *
       5.730383437168 86E-10)
  3.4028235E38 = maxBoost
 1.0 = queryBoost
)
```

```
--------------------------------------------------------------------------------
2 Hit: score(2.8052164E-4) | receivedDate(21.09.2015 03-23) | name(Senior
    Software Developer (Java / Backend) - User Management (m/w)Idealo
    Internet GmbH - Berlin) | id(56726eb344aeeb036a01c165)
2 Explain(: 2.8052164E-4 = (MATCH) function score, product of:
 0.4031614 = (MATCH) sum of:
   0.17038727 = (MATCH) sum of:
     0.103017226 = (MATCH) weight(content:softwareentwickl in 1222)
         [PerFieldSimilarity], result of:
       0.103017226 = score(doc=1222,freq=1.0), product of:
         0.50549334 = queryWeight, product of:
           3.7265449 = idf(docFreq=149, maxDocs=2292)
           0.13564666 = queryNorm
         0.20379542 = fieldWeight in 1222, product of:
           1.0 = tf(freq=1.0), with freq of:
             1.0 = termFreq=1.0
           3.7265449 = idf(docFreq=149, maxDocs=2292)
           0.0546875 = fieldNorm(doc=1222)
     0.06737005 = (MATCH) weight(content:java in 1222)
         [PerFieldSimilarity], result of:
       0.06737005 = score(doc=1222,freq=1.0), product of:
         0.40878406 = queryWeight, product of:
           3.0135949 = idf(docFreq=305, maxDocs=2292)
           0.13564666 = queryNorm
         0.16480596 = fieldWeight in 1222, product of:
           1.0 = tf(freq=1.0), with freq of:
             1.0 = termFreq=1.0
           3.0135949 = idf(docFreq=305, maxDocs=2292)
           0.0546875 = fieldNorm(doc=1222)
   0.23277412 = (MATCH) weight(content:testautomatisierung in 1222)
       [PerFieldSimilarity], result of:
     0.23277412 = score(doc=1222,freq=1.0), product of:
       0.75984997 = queryWeight, product of:
         5.601686 = idf(docFreq=22, maxDocs=2292)
         0.13564666 = queryNorm
       0.3063422 = fieldWeight in 1222, product of:
         1.0 = tf(freq=1.0), with freq of:
           1.0 = termFreq=1.0
         5.601686 = idf(docFreq=22, maxDocs=2292)
         0.0546875 = fieldNorm(doc=1222)
 6.958048E-4 = (MATCH) Math.min of
   6.958048E-4 = (MATCH) Function for field receivedDate:
     6.958048E-4 = exp(- MIN[Math.max(Math.abs(1.44279863E12(=doc value) -
         1.455486160446E12(=origin))) - 0.0(=offset), 0)] *
         5.73038343716886E-10)
   3.4028235E38 = maxBoost
 1.0 = queryBoost
)
--------------------------------------------------------------------------------
```

```
3 Hit: score(2.8052164E-4) | receivedDate(21.09.2015 03-23) | name(Senior
    Software Developer (Java) Single Sign-On) | id(567276e244aeeb036a01c3d0)
3 Explain(: 2.8052164E-4 = (MATCH) function score, product of:
 0.4031614 = (MATCH) sum of:
  0.17038727 = (MATCH) sum of:
   0.103017226 = (MATCH) weight(content:softwareentwickl in 1840)
       [PerFieldSimilarity], result of:
     0.103017226 = score(doc=1840,freq=1.0), product of:
      0.50549334 = queryWeight, product of:
       3.7265449 = idf(docFreq=149, maxDocs=2292)
       0.13564666 = queryNorm
      0.20379542 = fieldWeight in 1840, product of:
       1.0 = tf(freq=1.0), with freq of:
        1.0 = termFreq=1.0
       3.7265449 = idf(docFreq=149, maxDocs=2292)
       0.0546875 = fieldNorm(doc=1840)
   0.06737005 = (MATCH) weight(content:java in 1840)
       [PerFieldSimilarity], result of:
     0.06737005 = score(doc=1840,freq=1.0), product of:
      0.40878406 = queryWeight, product of:
       3.0135949 = idf(docFreq=305, maxDocs=2292)
       0.13564666 = queryNorm
      0.16480596 = fieldWeight in 1840, product of:
       1.0 = tf(freq=1.0), with freq of:
        1.0 = termFreq=1.0
       3.0135949 = idf(docFreq=305, maxDocs=2292)
       0.0546875 = fieldNorm(doc=1840)
  0.23277412 = (MATCH) weight(content:testautomatisierung in 1840)
      [PerFieldSimilarity], result of:
    0.23277412 = score(doc=1840,freq=1.0), product of:
     0.75984997 = queryWeight, product of:
      5.601686 = idf(docFreq=22, maxDocs=2292)
      0.13564666 = queryNorm
     0.3063422 = fieldWeight in 1840, product of:
      1.0 = tf(freq=1.0), with freq of:
       1.0 = termFreq=1.0
      5.601686 = idf(docFreq=22, maxDocs=2292)
      0.0546875 = fieldNorm(doc=1840)
 6.958048E-4 = (MATCH) Math.min of
  6.958048E-4 = (MATCH) Function for field receivedDate:
   6.958048E-4 = exp(- MIN[Math.max(Math.abs(1.44279863E12(=doc value) -
       1.455486160446E12(=origin))) - 0.0(=offset), 0)] *
       5.730383437168 86E-10)
  3.4028235E38 = maxBoost
 1.0 = queryBoost
)
```

Anhang 5.3: Query Nr. 4 - Explain-Plan des ersten Hit

```
1 Hit: score(6.45385E-4) | receivedDate(01.10.2015 02-55) | name(Senior
    Java-Entwickler Angebotsimport (m/w)Idealo Internet GmbH - Berlin) |
    id(567264c744aeeb036a01be31)
1 Explain(: 6.45385E-4 = (MATCH) function score, product of:
  0.56651473 = (MATCH) sum of:
    0.13950714 = (MATCH) weight(read:F in 403) [PerFieldSimilarity], result
       of:
      0.13950714 = score(doc=403,freq=1.0), product of:
        0.1369141 = queryWeight, product of:
          1.0189391 = idf(docFreq=2248, maxDocs=2292)
          0.13436927 = queryNorm
        1.0189391 = fieldWeight in 403, product of:
          1.0 = tf(freq=1.0), with freq of:
            1.0 = termFreq=1.0
          1.0189391 = idf(docFreq=2248, maxDocs=2292)
          1.0 = fieldNorm(doc=403)
    0.19642553 = (MATCH) sum of:
      0.102047116 = (MATCH) weight(content:softwareentwickl in 403)
          [PerFieldSimilarity], result of:
        0.102047116 = score(doc=403,freq=1.0), product of:
          0.50073314 = queryWeight, product of:
            3.7265449 = idf(docFreq=149, maxDocs=2292)
            0.13436927 = queryNorm
          0.20379542 = fieldWeight in 403, product of:
            1.0 = tf(freq=1.0), with freq of:
              1.0 = termFreq=1.0
            3.7265449 = idf(docFreq=149, maxDocs=2292)
            0.0546875 = fieldNorm(doc=403)
      0.09437842 = (MATCH) weight(content:java in 403) [PerFieldSimilarity],
          result of:
        0.09437842 = score(doc=403,freq=2.0), product of:
          0.40493453 = queryWeight, product of:
            3.0135949 = idf(docFreq=305, maxDocs=2292)
            0.13436927 = queryNorm
          0.23307082 = fieldWeight in 403, product of:
            1.4142135 = tf(freq=2.0), with freq of:
              2.0 = termFreq=2.0
            3.0135949 = idf(docFreq=305, maxDocs=2292)
            0.0546875 = fieldNorm(doc=403)
    0.23058207 = (MATCH) weight(content:testautomatisierung in 403)
        [PerFieldSimilarity], result of:
      0.23058207 = score(doc=403,freq=1.0), product of:
        0.7526944 = queryWeight, product of:
          5.601686 = idf(docFreq=22, maxDocs=2292)
          0.13436927 = queryNorm
        0.3063422 = fieldWeight in 403, product of:
          1.0 = tf(freq=1.0), with freq of:
```

```
        1.0 = termFreq=1.0
       5.601686 = idf(docFreq=22, maxDocs=2292)
       0.0546875 = fieldNorm(doc=403)
 0.0011392202 = (MATCH) Math.min of
   0.0011392202 = (MATCH) Function for field receivedDate:
     0.0011392202 = exp(- MIN[Math.max(Math.abs(1.443660906E12(=doc value)
         - 1.455488057445E12(=origin))) - 0.0(=offset), 0)] *
         5.73038343716886E-10)
   3.4028235E38 = maxBoost
 1.0 = queryBoost
)
```

Anhang 5.4: Query Nr. 5 - Explain-Plan des ersten Hits

```
1 Hit: score(4.9056945E-4) | receivedDate(01.10.2015 02-55) | name(Senior
    Java-Entwickler Angebotsimport (m/w)Idealo Internet GmbH - Berlin) |
    id(567264c744aeeb036a01be31)
1 Explain(: 4.9056945E-4 = (MATCH) function score, product of:
 0.431067 = (MATCH) sum of:
  0.19829285 = (MATCH) sum of:
   0.103017226 = (MATCH) weight(content:softwareentwickl in 403)
       [PerFieldSimilarity], result of:
     0.103017226 = score(doc=403,freq=1.0), product of:
      0.50549334 = queryWeight, product of:
       3.7265449 = idf(docFreq=149, maxDocs=2292)
       0.13564666 = queryNorm
      0.20379542 = fieldWeight in 403, product of:
       1.0 = tf(freq=1.0), with freq of:
        1.0 = termFreq=1.0
       3.7265449 = idf(docFreq=149, maxDocs=2292)
       0.0546875 = fieldNorm(doc=403)
   0.09527563 = (MATCH) weight(content:java in 403) [PerFieldSimilarity],
       result of:
     0.09527563 = score(doc=403,freq=2.0), product of:
      0.40878406 = queryWeight, product of:
       3.0135949 = idf(docFreq=305, maxDocs=2292)
       0.13564666 = queryNorm
      0.23307082 = fieldWeight in 403, product of:
       1.4142135 = tf(freq=2.0), with freq of:
        2.0 = termFreq=2.0
       3.0135949 = idf(docFreq=305, maxDocs=2292)
       0.0546875 = fieldNorm(doc=403)
  0.23277412 = (MATCH) weight(content:testautomatisierung in 403)
      [PerFieldSimilarity], result of:
    0.23277412 = score(doc=403,freq=1.0), product of:
     0.75984997 = queryWeight, product of:
      5.601686 = idf(docFreq=22, maxDocs=2292)
      0.13564666 = queryNorm
     0.3063422 = fieldWeight in 403, product of:
      1.0 = tf(freq=1.0), with freq of:
       1.0 = termFreq=1.0
      5.601686 = idf(docFreq=22, maxDocs=2292)
      0.0546875 = fieldNorm(doc=403)
 0.0011380353 = (MATCH) Math.min of
  0.0011380353 = (MATCH) Function for field receivedDate:
   0.0011380353 = exp(- MIN[Math.max(Math.abs(1.443660906E12(=doc value)
       - 1.455489873391E12(=origin))) - 0.0(=offset), 0)] *
       5.730383437168 86E-10)
  3.4028235E38 = maxBoost
 1.0 = queryBoost
)
```

Anhang 5.5: Query Nr. 6

```
{
  "function_score": {
    "query": {
      "filtered": {
        "query": {
          "bool": {
            "should": [
              {
                "match": {
                  "content": {
                    "query": "Spring Framework",
                    "type": "boolean",
                    "operator": "AND",
                    "boost": 50.0
                  }}
              },
              {
                "match": {
                  "content": {
                    "query": "Java",
                    "type": "boolean",
                    "operator": "AND",
                    "boost": 30.0
                  }}
              },
              {
                "match": {
                  "content": {
                    "query": "PostgreSQL",
                    "type": "boolean",
                    "operator": "AND",
                    "boost": 20.0
                  }}
              },
              {
                "match": {
                  "content": {
                    "query": "Datenbanken",
                    "type": "boolean",
                    "operator": "AND",
                    "boost": 20.0
                  }}}
            ]}
        },
        "filter": {
          "term": {
            "read": false
```

```
        }}}
    },
    "functions": [
      {
        "exp": {
          "receivedDate": {
            "scale": "14d"
          }}}
    ]}
}
```

Anhang 5.6: Query Nr. 6 - Explain-Plan von ersten Hit

```
1 Hit: score(5.763883E-4) | receivedDate(01.10.2015 02-55) | name(Senior
    Java-Entwickler Angebotsimport (m/w)Idealo Internet GmbH - Berlin) |
    id(567264c744aeeb036a01be31)
1 Explain(: 5.763883E-4 = (MATCH) function score, product of:
  0.39150882 = (MATCH) product of:
    0.52201176 = (MATCH) sum of:
      0.39096564 = (MATCH) sum of:
        0.17041172 = (MATCH) weight(content:spring in 403)
            [PerFieldSimilarity], result of:
          0.17041172 = score(doc=403,freq=2.0), product of:
            0.5933848 = queryWeight, product of:
              3.7132995 = idf(docFreq=151, maxDocs=2292)
              0.15979987 = queryNorm
            0.28718585 = fieldWeight in 403, product of:
              1.4142135 = tf(freq=2.0), with freq of:
                2.0 = termFreq=2.0
              3.7132995 = idf(docFreq=151, maxDocs=2292)
              0.0546875 = fieldNorm(doc=403)
        0.22055392 = (MATCH) weight(content:framework in 403)
            [PerFieldSimilarity], result of:
          0.22055392 = score(doc=403,freq=3.0), product of:
            0.609988 = queryWeight, product of:
              3.8171992 = idf(docFreq=136, maxDocs=2292)
              0.15979987 = queryNorm
            0.36157092 = fieldWeight in 403, product of:
              1.7320508 = tf(freq=3.0), with freq of:
                3.0 = termFreq=3.0
              3.8171992 = idf(docFreq=136, maxDocs=2292)
              0.0546875 = fieldNorm(doc=403)
      0.06734424 = (MATCH) weight(content:java^30.0 in 403)
          [PerFieldSimilarity], result of:
        0.06734424 = score(doc=403,freq=2.0), product of:
          0.28894326 = queryWeight, product of:
            30.0 = boost
            3.0135949 = idf(docFreq=305, maxDocs=2292)
            0.0031959976 = queryNorm
```

```
        0.23307082 = fieldWeight in 403, product of:
          1.4142135 = tf(freq=2.0), with freq of:
            2.0 = termFreq=2.0
          3.0135949 = idf(docFreq=305, maxDocs=2292)
          0.0546875 = fieldNorm(doc=403)
      0.06370189 = (MATCH) weight(content:datenbank^20.0 in 403)
          [PerFieldSimilarity], result of:
        0.06370189 = score(doc=403,freq=2.0), product of:
          0.22945254 = queryWeight, product of:
            20.0 = boost
            3.5896857 = idf(docFreq=171, maxDocs=2292)
            0.0031959976 = queryNorm
          0.27762556 = fieldWeight in 403, product of:
            1.4142135 = tf(freq=2.0), with freq of:
              2.0 = termFreq=2.0
            3.5896857 = idf(docFreq=171, maxDocs=2292)
            0.0546875 = fieldNorm(doc=403)
    0.75 = coord(3/4)
   0.0014722231 = (MATCH) Math.min of
    0.0014722231 = (MATCH) Function for field receivedDate:
      0.0014722231 = exp(- MIN[Math.max(Math.abs(1.443660906E12(=doc value)
          - 1.455040566297E12(=origin))) - 0.0(=offset), 0)] *
          5.730383437168 86E-10)
    3.4028235E38 = maxBoost
   1.0 = queryBoost
)
```

Anhang 5.7: Query Nr. 7

```
{
  "function_score": {
    "query": {
      "filtered": {
        "query": {
          "bool": {
            "should": [
              {
                "function_score": {
                  "query": {
                    "match": {
                      "content": {
                        "query": "Spring Framework",
                        "type": "boolean",
                        "operator": "OR",
                        "boost": 50.0
                      }}
                  },
                  "functions": [
                    {
                      "weight": 50.0
                    }
                  ]}
              },
              {
                "function_score": {
                  "query": {
                    "match": {
                      "content": {
                        "query": "Java",
                        "type": "boolean",
                        "operator": "OR",
                        "boost": 30.0
                      }}
                  },
                  "functions": [
                    {
                      "weight": 30.0
                    }
                  ]}
              },
              {
                "function_score": {
                  "query": {
                    "match": {
                      "content": {
                        "query": "PostgreSQL",
```

```
                    "type": "boolean",
                    "operator": "OR",
                    "boost": 20.0
                  }}
              },
              "functions": [
                {
                  "weight": 20.0
                }
              ]}
            },
            {
              "function_score": {
                "query": {
                  "match": {
                    "content": {
                      "query": "Datenbanken",
                      "type": "boolean",
                      "operator": "OR",
                      "boost": 20.0
                    }
                  }
                },
                "functions": [
                  {
                    "weight": 20.0
                  }
                ]}}
          ]}
      },
      "filter": {
        "term": {
          "read": false
        }}}
  },
  "functions": [
    {
      "exp": {
        "receivedDate": {
          "scale": "14d"
        }}}
  ]}
}
```

Anhang 5.8: Query Nr. 7 - Explain-Plan von ersten Hit

```
1 Hit: score(0.024797508) | receivedDate(01.10.2015 02-55) | name(Senior
    Java-Entwickler Angebotsimport (m/w)Idealo Internet GmbH - Berlin) |
    id(567264c744aeeb036a01be31)
1 Explain(: 0.024797508 = (MATCH) function score, product of:
  17.131985 = (MATCH) product of:
    22.842648 = (MATCH) sum of:
      19.548283 = (MATCH) function score, product of:
        0.39096564 = (MATCH) sum of:
          0.17041172 = (MATCH) weight(content:spring in 403)
              [PerFieldSimilarity], result of:
            0.17041172 = score(doc=403,freq=2.0), product of:
              0.5933848 = queryWeight, product of:
                3.7132995 = idf(docFreq=151, maxDocs=2292)
                0.15979987 = queryNorm
              0.28718585 = fieldWeight in 403, product of:
                1.4142135 = tf(freq=2.0), with freq of:
                  2.0 = termFreq=2.0
                3.7132995 = idf(docFreq=151, maxDocs=2292)
                0.0546875 = fieldNorm(doc=403)
          0.22055392 = (MATCH) weight(content:framework in 403)
              [PerFieldSimilarity], result of:
            0.22055392 = score(doc=403,freq=3.0), product of:
              0.609988 = queryWeight, product of:
                3.8171992 = idf(docFreq=136, maxDocs=2292)
                0.15979987 = queryNorm
              0.36157092 = fieldWeight in 403, product of:
                1.7320508 = tf(freq=3.0), with freq of:
                  3.0 = termFreq=3.0
                3.8171992 = idf(docFreq=136, maxDocs=2292)
                0.0546875 = fieldNorm(doc=403)
        50.0 = (MATCH) Math.min of
          50.0 = (MATCH) product of:
            1.0 = constant score 1.0 - no function provided
            50.0 = weight
          3.4028235E38 = maxBoost
        1.0 = queryBoost
      2.0203273 = (MATCH) function score, product of:
        0.06734424 = (MATCH) weight(content:java^30.0 in 403)
            [PerFieldSimilarity], result of:
          0.06734424 = score(doc=403,freq=2.0), product of:
            0.28894326 = queryWeight, product of:
              30.0 = boost
              3.0135949 = idf(docFreq=305, maxDocs=2292)
              0.0031959976 = queryNorm
            0.23307082 = fieldWeight in 403, product of:
              1.4142135 = tf(freq=2.0), with freq of:
                2.0 = termFreq=2.0
```

```
          3.0135949 = idf(docFreq=305, maxDocs=2292)
          0.0546875 = fieldNorm(doc=403)
      30.0 = (MATCH) Math.min of
        30.0 = (MATCH) product of:
          1.0 = constant score 1.0 - no function provided
          30.0 = weight
        3.4028235E38 = maxBoost
      1.0 = queryBoost
    1.2740378 = (MATCH) function score, product of:
      0.06370189 = (MATCH) weight(content:datenbank^20.0 in 403)
          [PerFieldSimilarity], result of:
        0.06370189 = score(doc=403,freq=2.0), product of:
          0.22945254 = queryWeight, product of:
            20.0 = boost
            3.5896857 = idf(docFreq=171, maxDocs=2292)
            0.0031959976 = queryNorm
          0.27762556 = fieldWeight in 403, product of:
            1.4142135 = tf(freq=2.0), with freq of:
              2.0 = termFreq=2.0
            3.5896857 = idf(docFreq=171, maxDocs=2292)
            0.0546875 = fieldNorm(doc=403)
      20.0 = (MATCH) Math.min of
        20.0 = (MATCH) product of:
          1.0 = constant score 1.0 - no function provided
          20.0 = weight
        3.4028235E38 = maxBoost
      1.0 = queryBoost
  0.75 = coord(3/4)
 0.0014474393 = (MATCH) Math.min of
  0.0014474393 = (MATCH) Function for field receivedDate:
    0.0014474393 = exp(- MIN[Math.max(Math.abs(1.443660906E12(=doc value)
        - 1.455070193609E12(=origin))) - 0.0(=offset), 0)] *
        5.730383437168 86E-10)
  3.4028235E38 = maxBoost
 1.0 = queryBoost
)
```

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Datum: 3. März 2016, Berlin **Unterschrift:**